Development Of A Performance-Portable Framework For Atomistic Simulations

Saunders, William Robert

Award date:
2019

Awarding institution:
University of Bath

Development Of A Performance-Portable Framework For Atomistic Simulations

submitted by

William Robert Saunders

for the degree of Doctor of Philosophy

of the

University of Bath

Department of Mathematical Sciences

December 2018

COPYRIGHT

LICENSE

INCLUSION OF PUBLISHED MATERIAL

This thesis includes material from the following published articles:

W. R. SAUNDERS, J. GRANT AND E. H. MÜLLER, *A domain specific language for performance portable molecular dynamics algorithms*, Computer Physics Communications, 224 (2018), pp. 119 - 135, `https://doi.org/10.1016/j.cpc.2017.11.006`

W. R. SAUNDERS, J. GRANT AND E. H. MÜLLER, *Long Range Forces in a Performance Portable Molecular Dynamics Framework*, Parallel Computing is Everywhere, 2018, pp. 37 - 46, `http://doi.org/10.3233/978-1-61499-843-3-37`

SUMMARY

The simulation of atomic scale interactions is an important tool in the fields of computational physics and chemistry. These simulations model interactions between large numbers of individual particles to provide quantitative results that inform and guide physical experiments. Modelling large numbers of interacting particles is a computationally expensive process that accounts for a significant proportion of CPU time in high performance computing facilities. Furthermore, even with large parallel computers modern simulations cannot model the vast numbers of particles that exist in relatively small amounts of physical material. Hence there is significant motivation to design and implement algorithms which model particle systems in the most computationally efficient manner possible, this is a highly non-trivial task due to the diversity and complexity of modern high performance computing hardware. It is important to write simulation code which is performant and portable between computing hardware, to address these challenges this thesis makes the following contributions:

Technical Contributions

- We present a new mathematical abstraction in which algorithms involving interacting particles can be described. We demonstrate the abstraction by describing non-bonded interactions between particles and by describing two structure analysis techniques.

- We implemented an interface to our code generation framework in terms of our abstraction. This code generation framework generates efficient parallel code for two prevalent high performance computing architectures and we demonstrate that the generated code is competitive in comparison to well established libraries.

- We provide a parallel implementation of the Ewald summation method written in our abstraction. This Ewald implementation extends the capabilities of our framework to include long-range electrostatic interactions.

- We provide a parallel Fast Multipole Method (FMM) implementation to further extend the electrostatic capabilities of our framework. We demonstrate that this FMM implementation scales well in parallel and is performant when simulated systems contain millions of charged particles.

Publications And Presentations

I presented the following work in both oral and written form.

- Publication: W. R. SAUNDERS, J. GRANT AND E. H. MÜLLER, *A domain specific language for performance portable molecular dynamics algorithms*, Computer Physics Communications, 224 (2018), pp. 119 - 135, `https://doi.org/10.1016/j.cpc.2017.11.006`. Here we present the abstraction and interface to the code generation framework alongside results from simulations of non-bonded interactions and structure analysis techniques.

- Presentation and publication (ParCo 2017): W. R. SAUNDERS, J. GRANT AND E. H. MÜLLER, *Long Range Forces in a Performance Portable Molecular Dynamics Framework*, Parallel Computing is Everywhere, 2018, pp. 37 - 46, `http://doi.org/10.3233/978-1-61499-843-3-37`. Here we present our parallel Ewald summation implementation within our abstraction and code generation framework.

- Presentation (CIUK 2016): W. R. SAUNDERS, J. GRANT AND E. H. MÜLLER, *Performance portable molecular dynamics*. We presented the abstraction and the performance of the framework to a general high performance computing audience.

- Presentation (Firedrake 2017): W. R. SAUNDERS, J. GRANT AND E. H. MÜLLER, *Performance portable molecular dynamics*. We presented the abstraction and the parallel performance of the framework to an audience familiar with high-level abstractions and code generation frameworks to compute numerical solutions of partial differential equations.

- Presentations at the University of Bath: W. R. SAUNDERS, J. GRANT AND E. H. MÜLLER. We presented our abstraction and implementation at the HPC Symposium in years 2015-2018 and presented existing algorithms twice at the Dept. of Mathematical Sciences Numerical Analysis seminar.

Thesis Structure

In Chapter 1 we discuss the background material to motivate molecular dynamics simulations and review existing software approaches. Chapter 2 presents our separation of concerns based abstraction in which particle based algorithms can be described. In Chapter 3 we describe our code generation process to produce efficient and portable code using our abstraction as input and compare performance with existing libraries. Electrostatic interactions are a major component of molecular dynamics simulations and in Chapter 4 we discuss two algorithms to compute these interactions, we compare the performance of our implementations of these algorithms with existing codes in Chapter 5. Finally, in Chapter 6 we conclude this thesis and consider possible future directions.

ACKNOWLEDGEMENTS

Over the past 4 years my supervisors, Eike Müller, James Grant, Robert Scheichl and Steve Parker, have provided excellent support and expertise. I am very much aware that the standard of supervision I have been provided by my supervisors has been greater and more productive than that experienced by many of my fellow PhD students.

Something much less quantifiable is the support I have received directly and indirectly from the community of PhD students and postdocs and I wish you all the best with your current and future endeavours. Furthermore, I am very grateful to my friends and family for their support and understanding throughout my PhD and for keeping me sane.

Throughout this project, I have made extensive use of the University of Bath HPC facility *Balena*. I am very grateful to Steven Chapman and Roshan Mathew for their advice and support.

Finally, if you are reading this as a PhD student, my advice would be the following: with correct application of thought you should not be afraid of doing something wrong, but be prepared to correct the mistake if it does not work.

This PhD project was funded by EPSRC.

CONTENTS

List of Figures iv
List of Tables ix
List of Algorithms xi

1 Background **1**
1.1 Scientific Background 1
1.1.1 Introduction 1
1.1.2 Statistical Mechanics 3
1.1.3 Molecular Dynamics 6
1.1.4 Analysis Techniques 15
1.2 Modern High Performance Computing 19
1.2.1 General HPC Facility Topology 20
1.2.2 Compute Nodes 20
1.3 Discussion Of Existing Libraries 28
1.3.1 Overview Of Existing Libraries 28
1.3.2 Library Comparisons 29
1.3.3 Discussion And Conclusions 31

2 A Separation of Concerns Based Abstraction **32**
2.1 PyOP2 And Firedrake: An Existing Approach 32
2.2 An Abstraction For Particle Operations 35
2.3 Abstraction Implementation 37
2.3.1 Domain Specific Language 37
2.3.2 Further Examples 46

3 Code Generation Of Modern Parallel MD Algorithms **52**
3.1 Modern Parallel MD Algorithms 52
3.1.1 Cell Based Methods 52
3.1.2 Parallel Decomposition 54

3.1.3 Halo Exchange . . . 55
3.1.4 Cell To Particle Maps . . . 57
3.1.5 Finding And Storing Pairs Of Particles . . . 59
3.1.6 Neighbour List Rebuilding . . . 65
3.2 Code Generation . . . 65
3.2.1 Particle Loop . . . 67
3.2.2 Local Particle Pair Loop . . . 72
3.3 Results . . . 77
3.3.1 Comparison To Other Codes . . . 79
3.3.2 Structure Analysis Algorithms . . . 83

4 Modern Algorithms for Electrostatic Interactions 86
4.1 Introduction . . . 86
4.1.1 Coulomb Potential Truncation . . . 88
4.2 Particle Ewald Summation . . . 89
4.2.1 Parameter Selection . . . 96
4.3 Fast Multipole Method . . . 98
4.3.1 Two Dimensional Fast Multipole Method . . . 98
4.3.2 Three Dimensional Fast Multipole Method . . . 112

5 Implementation of Electrostatic Interaction Algorithms 126
5.1 Ewald Implementation . . . 126
5.2 Ewald Results . . . 130
5.2.1 Computational Complexity . . . 130
5.2.2 Strong Scaling . . . 131
5.3 Fast Multipole Method . . . 132
5.3.1 Indirect Interactions . . . 133
5.3.2 Direct Interactions . . . 136
5.4 Fast Multipole Method Results . . . 137

6 Conclusion And Future Work 142
6.1 Summary Of Work . . . 142
6.2 Critical Assessment And Future Work . . . 144

A Appendices 149
A.1 Largest Subcluster Algorithm . . . 149
A.2 Negative Binomial Expansion . . . 150
A.3 Gaussian Units . . . 150
A.4 3D FMM Force Calculation . . . 150
A.5 Balena System Architecture . . . 151
A.6 Example LAMMPS Input Script . . . 152
A.7 Example HOOMD-blue Python Input Script . . . 153

A.8 CUDA Code Generation 153
A.8.1 CUDA Particle Loop 153
A.8.2 CUDA Local Particle Pair Loop 160
A.9 Convergence Characteristics Of Ewald Summation 163

Bibliography **166**

LIST OF FIGURES

1-1 Lennard-Jones potential plotted in solid black, in dashed black the same potential truncated and shifted at $r_c/\sigma = 2^{1/6} \approx 1.1$ (dotted vertical line) to create a repulsive only interaction. 8

1-2 Two ellipsoidal particles with axes of symmetry $\vec{u}_1$ and $\vec{u}_2$ separated by $\vec{r}$. . 9

1-3 Common neighbour analysis for bonded atom pair (i, j) (empty circles). The set of common neighbours (filled circles) are classified as a $(4, 2, 1)$ triplet. 18

1-4 Example node configuration comprised of two CPUs each with 8 compute cores. Performance numbers for memory bandwidth, compute rate and network bandwidth are representative of an Intel E5-2650v2 (2.6 GHz Ivy Bridge) combined with an Intel TrueScale QDR network interface. 22

1-5 Example node configuration comprised of two CPUs each with 8 compute cores paired with some accelerator. Performance numbers are representative of an Intel E5-2650v2 (2.6 GHz Ivy Bridge) system combined with an Intel TrueScale network interface. 23

2-1 Triangle described by three edges and three vertices, adapted from PyOP2 documentation [69]. 33

2-2 Triangle from Figure 2-1 with positions added, adapted from PyOP2 documentation [69]. 33

2-3 Triangle from Figure 2-1 after translation along $(1, 1)$, adapted from PyOP2 documentation [69]. 34

2-4 Example of direct (left) and indirect (centre and right) bonds as described by the sets $\mathcal{E}_d^{(i)}$, $\overline{\mathcal{E}}^{(i)}$ and $\mathcal{E}^{(i)}$ in Equations (2.8) and (2.9). The bond (v, w) in the central diagram would be counted twice in $\overline{\mathcal{E}}^{(i)}$ but only once in $\mathcal{E}^{(i)}$. 48

2-5 Local bonds used for CNA construction 50

3-1 Left: Circle of radius r_c around a selected particle. Right: Corresponding cells (in 2D) that contain all potential neighbours within a radius r_c. 54
3-2 Decomposing a domain into four sub-domains. 54
3-3 Halo exchange that must occur between two horizontally adjacent sub-domains. 55
3-4 2D version of fully periodic halo exchange pattern. 4 sub-domains, each own a 2×2 grid of cells shown in grey. Numbers in cells indicate the sub-domain that owns data in the cell. Left: (north, south) exchange between sub-domains 0 and 2 indicated by arrows. Right: (east, west) exchange indicated between sub-domains 0 and 1. Note the second exchange (right) includes the data from the first (left) exchange. 56
3-5 Example of a cell list (right) constructed from a sub-domain with 4 cells (left). The arrows follow the path traced to retrieve the indices of particles in cell 2. 58
3-6 Example of H, l and k for a simple 4 cell sub-domain. 59
3-7 2D comparison between circle of radius r_c and cells inspected for neighbours. 60
3-8 Example of the neighbour list and associated starting points. 62
3-9 Strong scaling experiment: parallel speed-up (left) and parallel efficiency (right) for the time taken (s) to compute $n_{max} = 10^4$ Velocity Verlet iterations of $N = 10^6$ particles using DL_POLY, LAMMPS and our implementation (labeled as "Framework"). Efficiency and speed-up are relative to one full node (16 cores). Efficiency is calculated according to Equation (3.9). In the left plot perfect scaling is indicated by the dashed gray line. Raw results are presented in Table 3.3 and simulation parameters are tabulated in Table 3.2. 80
3-10 CPU-only weak scaling experiment: time taken to integrate the system over $n_{\max} = 5000$ time steps (left) and parallel efficiency (right). The efficiency relative to one full node (right) is calculated according to Equation (3.10). The top horizontal axes shows the total number N of particles in the system; the number of particles per core is kept fixed at $512,000$ ($8,192,000$ particles per node). 82
3-11 CPU-GPU weak scaling experiment with reduced particle number: time taken to simulate $n_{\max} = 5000$ time steps (left) parallel efficiency relative to a single GPU/node, calculated according to Equation (3.10) (right). The number of particles per node is kept fixed at 512,000. 82
3-12 Evolution of mean Q_4, Q_5 and Q_6 values over the course of the simulation. The horizontal dashed lines plot the expected Q_4 and Q_6 values of a perfect FCC lattice. 84

3-13 Probability density of Q_4 values (left) and Q_6 values (right) in final system configuration. (left) Dashed vertical line at $Q_4 = 0.097$ is the expected Q_4 value of a perfect hcp lattice. Dashed vertical line at $Q_4 = 0.191$ is the expected Q_4 value of a perfect fcc lattice. (right) Dashed vertical line at $Q_6 = 0.485$ is the expected Q_6 value of a perfect hcp lattice. Dashed vertical line at $Q_6 = 0.575$ is the expected Q_6 value of a perfect fcc lattice. 84
3-14 Weak scaling experiment that combines a simulation with on-the-fly analysis. Time taken to integrate 5000 steps, parallel efficiency relative to a single node (right). 85

4-1 Spherical system S of radius L and truncation radius $r_c = L - a$. 88
4-2 One dimensional representation of the charge splitting process for two positive charges and one negative charge. The $-\delta$,$-D^{(\mathrm{sr})}$ and $-D^{(\mathrm{lr})}$ labels the figure indicate the charge splitting process for the right-hand charge. 90
4-3 Log-scale plot of the short-range potential induced by a unit charge at the origin. 92
4-4 One dimension representation of the self interaction between charges and their corresponding long-range charge density. 94
4-5 A p-term multipole expansion at P constructed from charges contained in the square containing P will not give an accurate approximation of the potential φ in the shaded region. 100
4-6 Two well separated cells P and Q. 101
4-7 Four p-term multipole expansions at the points $P_1, \ldots, P_4$ are translated to the intersection point of the four cells. 101
4-8 The hierarchy of mesh levels for $\mathcal{L} = 3$. 105
4-9 Interaction list in grey for square i. Thick lines indicate the boundaries of the parent cells. 105
4-10 Overview of 2D FMM cost per step, adapted from [31]. 106
4-11 Overview of the flow of information to and from the grey square in a pass of the 2D FMM. Multipole to Multipole and Local to Local translations are illustrated with solid line arrows. Multipole to Local translations are denoted by dashed line arrows. For clarity multipole to local arrows are omitted on mesh level 3; the multipole source locations are indicated by "M" and the centre of local expansions by "L". The arrow denoted by "(*)" indicates a generic boundary condition method on level 0 that converts the p-term expansion $\Phi_{0,0}$ to $\Psi_{0,0}$ if applicable. 109
4-12 Simulation domain and periodic images. Central grey image represents the simulation domain, all other squares represent periodic images. The multipole expansion $\Phi_{0,0}$ centred in squares in the white region is not valid in the primary image. 110

4-13 Interaction list marked with crosses for square P. The grey square indicates the primary image and white squares indicate periodic images. Dotted lines indicate boundaries between the four squares per image. 111
4-14 Spherical coordinate convention. 112
4-15 Left: original multipole to local translation $\mathcal{T}_{\mathrm{ML}}$ along the vector (ρ, α, β) with $\mathcal{O}(p^4)$ computational complexity. Right: multipole to local translation performed by (1) rotating coordinate frame with operation $\mathcal{R}_y(\alpha)\mathcal{R}_z(\beta)$ (2) z-direction multipole to local translation $\mathcal{T}^z_{\mathrm{ML}}$ along new z-axis (3) rotate coefficients back into the original coordinate frame with operation $\mathcal{R}_z(-\beta)\mathcal{R}_y(-\alpha)$. 125

5-1 Time per iteration against particle count for an NaCl system on a single 8 core CPU using OpenMP (our framework) or pure MPI (DL_POLY_4). . . 131
5-2 Strong scaling experiment of an NaCl system comparing our implementation, labeled as "Framework", with DL_POLY_4. Time per iteration (left) and parallel efficiency relative to one 16-core node (right). Time taken is recorded for $3.3 \cdot 10^4$ charges over 300 Velocity Verlet iterations. Short-range Lennard-Jones interactions are enabled with a cutoff of $3\mathring{A}$. 132
5-3 2D OT distributed over 4 MPI ranks, MPI ranks are labeled in the centre of the cells. On level $l = 1$ all cells are owned by rank 0 to keep all the children of cell 0 on level 0 on the same MPI rank. 134
5-4 (Left) relative error in system potential energy from DL_POLY against input precision, dashed black arrows indicate the output error for an input precision of 10^{-6}. (Right) relative error in system potential energy and absolute error in particle forces against number of expansion terms used for all expansions, dashed arrows indicate the number of expansion terms required to meet the DL_POLY output error in the left plot. Average force error is computed as the average error over all charges over all component directions. Worst force error is computed as the maximum error over all ions and all component directions. 139
5-5 Strong scaling comparison between our FMM implementation, labeled as "PPMD", and DL_POLY FFT based Ewald. (Right) Time taken per Velocity Verlet iteration. (Left) Parallel efficiency as defined in Equation (3.9) computed relative to 1 node. One node consists of two Intel Xeon E5-2650v2 CPUs (16 cores per node). Time per iteration and parallel efficiency is recorded for a system containing 10^6 charges and a system containing $4 \cdot 10^6$ charges. 140

5-6 Weak scaling test of our FMM implementation. (Right) Time taken per Velocity Verlet iteration, floating numbers indicate the number of levels in the octal tree. (Left) Parallel efficiency as defined in Equation (3.10) computed relative to 1 node. One node consists of two Intel Xeon E5-2650v2 CPUs (16 cores per node). 141

A-1 LAMMPS Lennard-Jones example script from `http://lammps.sandia.gov/inputs/in.lj.txt` 152

A-2 HOOMD-blue Lennard-Jones example Python script from `http://glotzerlab.engin.umich.edu/hoomd-blue/doc/dump_dcd-example.html` 153

A-3 Two charges of strength q separated by distance d aligned parallel to the x-axis. 164

A-4 Long-range energy contribution of an approximate dipole system constructed from two charges separated by a distance d. Predicted long-range energy is plotted in dashed black, computed energy is plotted in solid black. Values are plotted for Gaussian width $\alpha = 1.0$ and reciprocal cutoff $k_c = 200$. . . . 165

LIST OF TABLES

1.1 Values of Q_4, Q_5 and Q_6 for perfect lattices, see [76] and Table 1 in [51]. . . 17

2.1 Fundamental data classes of the DSL 44
2.2 Supported access descriptors 45
2.3 Fundamental looping classes of the DSL 46

3.1 Access descriptors for the loops in the Velocity Verlet Algorithm 17. 78
3.2 Parameters of Lennard-Jones benchmark for the strong scaling experiment; units are chosen such that $\sigma = \epsilon = 1$ († = excluding DL_POLY, see main text). 79
3.3 Strong scaling experiment: time taken (s) to compute $n_{max} = 10^4$ Velocity Verlet iterations of $N = 10^6$ particles using DL_POLY, LAMMPS and our implementation (labeled as "Framework"). Further simulation parameters are given in Table 3.2. CPU nodes consist of two eight core E5-2650v2 CPUs, GPU nodes contain one or more K20X GPUs. GPUs are compared against CPU nodes on a one-to-one basis. For GPU results, the Framework used one CPU core as a host for each GPU. LAMMPS implements a GPU offload approach and hence used all CPU cores on the node in addition to available GPUs. All codes were built with the Intel 2016 compiler suite and OpenMPI 1.8.4 (with the exception of DL_POLY, which used OpenMPI 2.0.0). The NVIDIA CUDA toolkit version 7.5.18 was used for the GPU compilation and the framework was run with Python 2.7.8. 80
3.4 Absolute performance metrics (as percentage of peak performance and integration time) for two kernels recorded from GPU weak scaling experiment presented in Figure 3-11. The "Force & PE" kernel is only called every 10 iterations and hence accounts for a smaller proportion of the total runtime than the "Force" kernel. 83

3.5 Parameters of bond order analysis weak scaling experiment. Units are chose such that $\sigma = \epsilon = 1$. 85

A.1 Relevant differences between SI units and Gaussian units 150
A.2 *Balena* System Architecture 151

LIST OF ALGORITHMS

1 Time evolution using Velocity Verlet with time step δt. In this example we use an inter-particle force that is a function of particle positions, in general this force can be a function of all particle properties. 11

2 BOA Local Particle Pair Loop I. 47
3 BOA Particle Loop II. 48
4 CNA Local Particle Pair Loop I: Calculate direct bonds for each particle. . . 50
5 CNA Local Particle Pair Loop II: Calculate all other bonds in the local environment. 51
6 CNA Local Particle Pair Loop III: Calculate number of common neighbours $n_{\text{nb}}^{(i)}$, number of bonds $n_{\text{b}}^{(i)}$ between those common neighbours and the largest clustersize $n_{\text{lcb}}^{(i)}$. 51

7 Method to determine containing cell c_i of particle i. 53
8 Construction of linked list cell list. 57
9 Assigning layers to particles and determining cell occupancy counts. Rapaport [17] Section 3.4. 58
10 Propose pairs of particles by considering pairs of cells. 60
11 Construction of sequential neighbour list. 62
12 Pairwise kernel execution using a sequential neighbour list. 63
13 Construction of matrix neighbour list based on Rapaport [17] Section 3.4. . . 64
14 Interaction using matrix neighbour list. 65
15 Particle Loop code generation for `ParticleDats` 69
16 Particle Loop code generation for `ScalarArrays` and `GlobalArrays` 70
17 Velocity Verlet integrator used in Section 3.3. The system is integrated numerically with a time step of size δt until the final time $T = n_{\max}\delta t$. . . . 78

18 2D FMM algorithm to compute $\Psi_{l,i}$ for a system with free space boundary conditions. 107

19 2D FMM algorithm to compute the interactions between charges via p-term local expansions and direct charge-charge interactions. 108
20 3D FMM algorithm to compute $\Psi_{l,i}$ for a system with free space boundary conditions. 118
21 3D FMM algorithm to compute the interactions between charges via p-term local expansions and direct charge-charge interactions. 119

22 Computational kernel for the contribution to reciprocal space for a particle j. 129
23 Computational kernel to extract the long-range contribution from reciprocal space. 129
24 Cell by cell method to compute direct charge to charge interactions. 137

25 Calculate maximal cluster size. 149
26 Particle Loop code generation for `ParticleDats` 155
27 CUDA Particle Loop code generation for `ScalarArrays` and `GlobalArrays` . 156

CHAPTER 1

BACKGROUND

1.1 Scientific Background

1.1.1 Introduction

Computational Physics and Chemistry simulate the properties of materials on computers. Molecular Dynamics (MD) and Monte Carlo (MC) are the two main approaches that provide quantitative outputs that can be compared with experimental results. Computed results may complement experimental measurements. For example, one might want to find an optimal material contained within a huge family of materials. By using simulation techniques a set of candidate materials can be selected from a huge family of materials for physical experimentation without requiring time in a laboratory. Furthermore, computer simulation allows scientific investigation in regions of parameter space which are experimentally inaccessible, e.g. the high temperatures and pressures in solar plasmas or systems involving hazardous radioactive materials. In these cases, the cost of time and physical resource in a laboratory is significantly more than for computer simulation, furthermore, simulation may be the only viable option.

Due to their broad applications, both MD and MC simulations have become a major tool in computational physics, chemistry, biochemistry and drug discovery and account for a significant proportion of runtime on high performance computing (HPC) systems. As MD simulation is a commonly used technique across a wide range of disciplines we often refer to the collective group of users as "domain specialists" irrespective of scientific discipline.

This thesis focuses on the MD approach to simulating materials, however, the methods we describe are applicable to MC, for example, the Hybrid Monte Carlo scheme in section 1.1.3. In the MD approach physical objects, such as atoms and ions or potentially entire molecules, are represented by points in the simulation domain often referred to as particles. The particles interact and move through the simulation domain following Newton's laws of motion [53]. In classical physics a particle occupies some point in space known as its

position and carries a velocity that determines the rate of change of position. The set of positions and velocities taken by the particle are known as the trajectory of the particle.

The computationally expensive and interesting components are calculating the interactions between individual particles, which determine the force exerted on each particle and the potential energy of the whole system. Interactions are typically described by potentials which are both mathematically complex and computationally expensive. By modelling these interactions the equation of state can be predicted and system properties can be calculated from measurable quantities.

Properties of the simulated system are observed and quantified through a wide range of analysis techniques which extract and process information from the system of particles. Analysis techniques can exceed the computational complexity of the simulation itself. Furthermore, without careful implementation modern analysis can easily become prohibitively expensive to conduct, implementing high performance code for analysis may not be in the skill set of computational physicists and chemists.

Even with an efficient implementation the cost of computing interactions will dominate the overall cost of a typical simulation and impose a limit on the number of particles that a simulation can contain for a given computational resource. With a significant computational resource and an efficient code a state-of-the-art simulation may contain in the region of billions of particles [44, 1, 3], which is an extremely small number in comparison to the number of atoms in a gramme of material.

It is known that there are $\approx 6 \times 10^{23}$ atoms contained in 12 grammes of Carbon-12. Hence in real experiments that are conducted on multiple grammes of material there are a vast number of atoms involved, this indicates that experimentally observed properties are a macroscopic average over a large number of individual particles. Secondly, per particle events occur on microscopic timescales (femto-picoseconds) in contrast to experimentally observable properties that occur on much longer timescales.

Simulations are conducted with large numbers of particles (10^4 - 10^9) as the error of a statistical average is expected to decrease proportional to the reciprocal square root of the number of particles. Physical properties of a system may depend on the effective size of the system, if too few particles are simulated then observable quantities may be negatively impacted by these so called finite size effects. Typically, particle counts of $10^4 - 10^9$ are sufficient to avoid finite size effects, depending on the exact system.

In a computer simulation per particle properties, such as position and velocity, are modelled and per particle properties, such as potential energy, can be computed. Physically observable properties are calculated as ensemble averages, these provide a method to compute the macroscopic effect of the collection of particles over a period of time. Statistical mechanics describes how to estimate ensemble averages using the trajectories of all individual particles, these averages are ideally computed over long trajectories of a large number of particles, which is a computationally expensive process.

1.1.2 Statistical Mechanics

We now provide an introduction to statistical mechanics to motivate the use of Molecular Dynamics (MD) as a computational tool. More complete descriptions are given by Allen and Tildesley [2] and by Frenkel and Smit [24]. We begin by introducing the idea of the "state" of a system; a state is an instance of the degrees of freedom associated with a system. For example, if we consider a set of N non-interacting particles represented by points in $\mathbb{R}^3$ with some set of velocities also in $\mathbb{R}^3$ then there are $6N$ degrees of freedom. These $6N$ degrees of freedom describe the phase space of the system and we refer to each point $X \in \mathbb{R}^{6N}$ in the phase space as a state of the system. Since each set of positions and velocities that are produced by the time evolution of the system corresponds to a state in the phase space, the state X moves through phase space as the system evolves. The set of states in phase space that are visited by the time evolution of the system $\{X(t)|t \in [0, t_{end}]\}$ form a trajectory through the phase space. For simplicity we consider phase space and time to be discrete, for a continuous system all summations over phase space should be replaced by integrals.

For each state in the phase space of a system we assume that the total energy $\mathcal{H}$ is defined as,

$$\mathcal{H} = \mathcal{U} + \mathcal{K}, \tag{1.1}$$

where $\mathcal{U}$ is the system potential energy and $\mathcal{K}$ is the system kinetic energy. We assume the total potential energy $\mathcal{U}$ of a system of N particles is given by

$$\mathcal{U} = \sum_{j=1}^{N} \left(U_j^{\text{external}} + \frac{1}{2} U_j^{\text{inter-particle}} \right), \tag{1.2}$$

where U_j^{external} is the potential energy of particle j in any external field and $U_j^{\text{inter-particle}}$ is the potential energy of particle j in the potential field induced by all other particles. The factor of a half accounts for the double counting of potential energy between pairs of particles. Point particles carry translational momentum and zero angular momentum. For N particles the kinetic energy $\mathcal{K}$ of the system is given by

$$\mathcal{K} = \sum_{j=1}^{N} \frac{\vec{p}_j{}^2}{2m_j}, \tag{1.3}$$

where $\vec{p}_j$ and m_j are the translational momentum and mass of particle j respectively.

Physical experiments are conducted in the real world where it is impossible to completely isolate the experiment from the surrounding environment. Hence throughout an experiment energy will be continuously exchanged with the environment to some extent. A simulated system can either be perfectly isolated or coupled to a environment in a manner that allows energy to be exchanged between the simulation and the environment. In a MD simulation, where the evolution of the system is determined by integrating Newton's equation of motion, it is relatively straightforward to keep the total energy of the system

constant up to small fluctuations.

The constant energy scenario corresponds to an experiment which is completely isolated from the surroundings. However, physical experiments are often conducted at a constant temperature and pressure and it is reasonable to want to replicate these conditions within a simulation such that results can be more easily compared.

We define an ensemble to be the combination of a phase space and a probability density function (PDF) defined on the phase space. The PDF describes the probability of occurrence of each state in the phase space. More generally, statistical ensembles describe which thermodynamic properties are fixed throughout a simulation, for example, in the micro-canonical ensemble the number of particles N, simulation volume V and total energy E are fixed. In the canonical ensemble the number of particles N, simulation volume V and temperature T are fixed. The PDFs for these two ensembles are given by

$$P(X) = \begin{cases} Z_E^{-1}\delta(\mathcal{H}(X) - E) & \text{micro-canonical ensemble with energy } E \\ Z_T^{-1}\exp\left(\frac{-\mathcal{H}(X)}{k_B T}\right) & \text{canonical ensemble at temperature } T \end{cases} \tag{1.4}$$

where $Z_{\{E,T\}}$ are normalisation constants, $k_B \approx 1.38 \times 10^{-23}$ JK^{-1} is the Boltzmann constant and $\mathcal{H}(X)$ is the total energy of state X. The canonical ensemble describes a finite system thermally coupled to a infinite heat bath of temperature T, the PDF of the canonical ensemble is known as the Boltzmann distribution [25],

$$P(X) = \frac{\exp\left(\frac{-\mathcal{H}(X)}{k_B T}\right)}{\sum_Y \exp\left(\frac{-\mathcal{H}(Y)}{k_B T}\right)}. \tag{1.5}$$

A macroscopic property $\mathcal{A}$, such as pressure or potential energy, is a function of the system state $\mathcal{A} = \mathcal{A}(X)$. The ensemble average $\langle\mathcal{A}\rangle_{\text{ens}}$ of a property $\mathcal{A}$ is defined as the mean value of the property over all possible states in the ensemble,

$$\langle A\rangle_{\text{ens}} = \sum_X \mathcal{A}(X)P(X), \tag{1.6}$$

where $P(X)$ is the probability of state X in the ensemble. We wish to compute ensemble averages $\langle A\rangle_{\text{ens}}$ as with the correct choice of ensemble these should be comparable to experimental values.

In general, the normalisation constants Z, such as the denominator of Equation (1.5), have no known analytic solutions and are essentially impossible to compute exactly. The idea of MD and MC is to apply importance sampling to estimate the values of $\langle\mathcal{A}\rangle_{\text{ens}}$. We require that an algorithm satisfies two conditions, which together are sufficient to sample states according to the correct ensemble distribution, namely detailed balance and ergodicity. The transition probability π of the underlying Markov process satisfies detailed balance if

$$P(X)\pi(X \to X') = P(X')\pi(X' \to X), \tag{1.7}$$

where $\pi(X \to X')$ is the transition probability from X to X'. This thesis is focused on the MD approach, and we assume that the MD algorithms we describe satisfy detailed balance.

The second condition is ergodicity, we define an ergodic algorithm to be one where all points X in phase space with non-zero probability are reachable by the simulation. For example, in the constant energy micro-canonical ensemble at energy E the points in phase space with non-zero probability are those with energy E, for the simulation to be ergodic all states with energy E must be reachable. Furthermore, to satisfy this particular ensemble all valid points must occur with equal probability.

In the micro-canonical ensemble consider a system and corresponding phase space where there are disjoint surfaces of energy E, this situation occurs when a system contains some form of energy barrier. In the ensemble, for a trajectory to be ergodic it must be able to reach all points on all energy surfaces with energy E with equal probability. MD simulations that follow Newton's Laws of motion do not produce trajectories capable of "jumping" between energy surfaces and hence are not ergodic in the presence of energy barriers, Hybrid Monte Carlo is a technique that addresses this limitation.

Discrete trajectories $\mathcal{T} = \{X(t)|t \in \{\delta t, 2\delta t, \ldots, t_{\text{end}}\}\}$ are produced by both MD and MC simulations. If we assume a trajectory is produced in a manner that satisfies detailed balance and ergodicity then the states $X \in \mathcal{T}$ are distributed according to the ensemble distribution. The average value of a property $\mathcal{A}$ along a trajectory $\mathcal{T}$ is given by

$$\langle \mathcal{A} \rangle_{\mathcal{T}} = \frac{1}{N_{\text{steps}}} \sum_{\tau=1}^{N_{\text{steps}}} \mathcal{A}\left(\hat{X}(\tau \delta t)\right). \tag{1.8}$$

We assume that the average value of a property $\langle \mathcal{A} \rangle_{\mathcal{T}}$ along an ergodic trajectory $\mathcal{T}$ tends to the ensemble average $\langle \mathcal{A} \rangle_{\text{ens}}$ i.e.

$$\lim_{t_{\text{end}} \to \infty} \langle \mathcal{A} \rangle_{\mathcal{T}} = \langle \mathcal{A} \rangle_{\text{ens}}. \tag{1.9}$$

Hence in a MD or MC simulation we wish to produce long ergodic trajectories to produce good estimates of ensemble averages.

MC and MD are both iterative approaches to generate a trajectory through phase space, but these two techniques operate using fundamentally different approaches. In the classical MD approach each particle in the system is modelled as a point-wise object with a position, velocity and force. The force on each particle is computed using phenomenological potentials that describe the interactions with all other particles. Once the force on a particle is computed the trajectory of the particle is updated following Newton's Laws of motion.

In MC a new state is proposed by taking the current state and applying some perturbation. Firstly, if the proposed state is not a member of the ensemble it is rejected immediately, if the proposed state is a member of the ensemble it is accepted with a prob-

ability dependent on the energy difference. For example, a simple MC approach based on Metropolis Hastings [34] for the canonical ensemble takes a state with N particle positions $\vec{r}_N$ and potential energy $U = \mathcal{U}(\vec{r}_N)$ and proposes a new state with positions $\vec{r}'_N$ and potential energy $U' = \mathcal{U}(\vec{r}'_N)$. The acceptance probability of the proposed positions $\vec{r}'_N$ is given by $\min\left[1, \exp(U'/(k_B T))/\exp(U/(k_B T))\right]$.

1.1.3 Molecular Dynamics

The construction of a MD simulation involves multiple design parameters that are altered to suit the specific requirements of the simulation. We provide a description of a generic simulation configuration that is widely used for general simulations. We begin with the simulation domain which, in our configuration, is a cuboid with extents L_x, L_y, L_z. Our convention will be to place the origin of the coordinate system at the centre of the cuboid, a particle position $\vec{r}$ is valid if $\vec{r} \in [-L_x/2, L_x/2] \times [-L_y/2, L_y/2] \times [-L_z/2, L_z/2]$.

Particular attention needs to given to the boundary conditions of the simulation domain. We apply the convention of using periodic boundary conditions on all three dimensions of the cuboid. In simulations that are focused on simulating thin slab-like sections of material it may be sensible to only apply periodic boundaries in two of the three dimensions. The main idea of using periodic boundary conditions is to avoid introducing artificial surfaces [2] into a simulation that is focused on bulk material. Furthermore, it is a non-trivial exercise to construct boundary conditions that are not periodic and behave in a physically realistic manner as for large volumes of physical material the boundary becomes irrelevant.

Within the simulation domain N particles are initialised such that particle i has a position $\vec{r}_i$ and an initial momentum $\vec{v}_i m_i$ where $\vec{v}_i$ and m_i are the velocity and mass of the particle. The initial positions and velocities of particles are chosen to be physically sensible, for example, it would be unrealistic to begin a simulation with overlapping particles. The time evolution of the system is governed by coupled form of Newton's Second Law [53],

$$m_i \frac{\partial^2 \vec{r}_i}{\partial t^2} = \vec{F}_i, \ i = 1, \ldots, N, \tag{1.10}$$

where $\vec{F}_i$ is the force exerted on particle i from all other particles in the system plus any external field. Note that $\vec{F}_i$ is in principle a function of all particles in the system. In the general case, there is a contribution to $\vec{F}_i$ from the $N-1$ remaining particles located in the primary image and the N particles located in each periodic image of the simulation domain. If we write the force exerted on particle i by particle j in periodic image $\mathfrak{n} \in \mathbb{Z}^3$ as $\vec{F}_{ij,\mathfrak{n}}$ then in the absence of any external force field we can explicitly write the force $\vec{F}_i$ as

$$\vec{F}_i = \sum_{\mathfrak{n} \in \mathbb{Z}^3 \setminus \{\vec{0}\}} \sum_{j=1}^{N} \vec{F}_{ij,\mathfrak{n}} + \sum_{j \in \{1,\ldots,N\} \setminus \{i\}} \vec{F}_{ij,\vec{0}}. \tag{1.11}$$

We split Equation (1.11) into two terms to explicitly consider contributions from particles

in the primary image ($\mathfrak{n} = \vec{0}$) where the summation must exclude the $\vec{F}_{ii,\vec{0}}$ term that corresponds to particle self-interaction.

We will discuss two major inter-particle interaction types, the first describes short-range non-bonded interactions, such as Van der Waals forces, and the second describes long-range electrostatic interactions that exist in simulations containing charged particles. If two particles i and j interact via both a short-range component $\vec{F}^{\text{sr}}_{ij,\mathfrak{n}}$ and a long-range component $\vec{F}^{\text{lr}}_{ij,\mathfrak{n}}$ then the force $\vec{F}_i$ is given as the sum of forces from these two components,

$$\vec{F}_{ij,\mathfrak{n}} = \vec{F}^{\text{sr}}_{ij,\mathfrak{n}} + \vec{F}^{\text{lr}}_{ij,\mathfrak{n}}. \tag{1.12}$$

In this thesis we only consider $\vec{F}^{\text{lr}}_{ij,\mathfrak{n}}$ to be the long-range forces that exist from electrostatic interactions in systems of charged particles and a detailed discussion of these interactions is given in Chapter 4. We refer to electrostatic interactions as "long-range" as the effect of the interaction spans the entire simulation i.e. $\vec{F}^{\text{lr}}_{ij,\mathfrak{n}}$ cannot be truncated to zero at any cutoff radius without incurring an unbounded error as we demonstrate in Section 4.1.1.

Long-range interactions, in contrary to short-range interactions, are those with a functional form that can be safely truncated to zero for sufficiently separated particles, i.e. if a pair of particles are separated by a distance greater than some value r_c then in the simulation they exert no force on each other.

The ability to truncate interactions without incurring a significant error penalty is a key ingredient of competitive MD simulation codes, as with truncation the short-range interactions between N particles can be computed with $\mathcal{O}(N)$ computational complexity.

Often the short-range interaction between a pair of particles is described as a pairwise potential $U(r)$ that is a function of the inter-particle distance $r = |\vec{r}_j - \vec{r}_i|$ between the two particles i and j. The potential energy between i and j is computed by evaluating $U(|\vec{r}_j - \vec{r}_i|)$ and the magnitude of the force exerted on both particles is

$$F_{ij} = \left.\frac{-\partial U(r)}{\partial r}\right|_{r=|\vec{r}_j - \vec{r}_i|}. \tag{1.13}$$

Furthermore, the force exerted on particle i by particle j is computed by scaling the unit vector from particle i to particle j by the magnitude of the force:

$$\vec{F}_{ij} = -\frac{\vec{r}_j - \vec{r}_i}{|\vec{r}_j - \vec{r}_i|} \left.\frac{\partial U(r)}{\partial r}\right|_{r=|\vec{r}_j - \vec{r}_i|}. \tag{1.14}$$

A very common short-range potential is the Lennard-Jones [47] potential,

$$U_{\text{LJ}}(r) = 4\epsilon \left[\left(\frac{\sigma}{r}\right)^{12} - \left(\frac{\sigma}{r}\right)^{6} \right], \tag{1.15}$$

where ϵ and σ are parameters that are chosen to best approximate the true interaction that is being modelled. For this potential a pair of particles will be attracted to each other

if they are separated by a distance $r > 2^{1/6}\sigma$ and will repel each other if separated by a distance $r < 2^{1/6}\sigma$. As evaluating the potential $U_{\mathrm{LJ}}(r)$ gives a value that is interpreted as a potential energy ϵ must have units of energy and σ must have units of length.

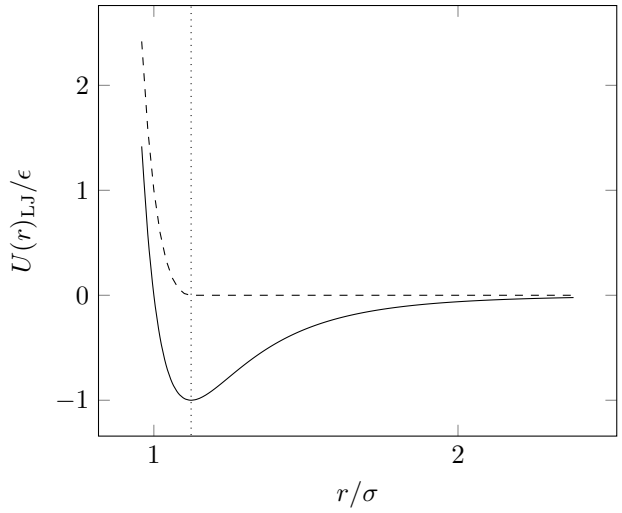

Figure 1-1: *Lennard-Jones potential plotted in solid black, in dashed black the same potential truncated and shifted at $r_c/\sigma = 2^{1/6} \approx 1.1$ (dotted vertical line) to create a repulsive only interaction.*

For larger values of r, i.e. $r/\sigma > 2$, the dominant term in the Lennard-Jones potential has the form r^{-6}, which converges to zero rapidly enough that the potential can be truncated to zero at some inter-particle distance r_c. When a potential is truncated at a distance r_c it is common practice to shift the entire potential by $U(r_c)$ such that there is no discontinuity in the potential at $r = r_c$ as plotted in Figure 1-1.

If we consider a simulation with only short-range interactions and if we can truncate the potential to zero for pairs of particles that are separated by a distance greater than r_c then we can revise Equation (1.11) to be computationally feasible and evaluate to a well defined value,

$$\vec{F}_i = \sum_{\substack{j \text{ s.t. } |\vec{r}_j - \vec{r}_i| < r_c \\ \text{and } i \neq j}} \vec{F}_{ij}. \tag{1.16}$$

In a similar manner, the total potential energy $\mathcal{U}$ of the system can be written as the sum of the potential energies between all pairs of particles that interact with a non-zero potential,

$$\mathcal{U} = \frac{1}{2} \sum_{i=1}^{N} \sum_{\substack{j \text{ s.t. } |\vec{r}_j - \vec{r}_i| < r_c \\ \text{and } i \neq j}} U(|\vec{r}_j - \vec{r}_i|). \tag{1.17}$$

For a particle i at position $\vec{r}_i$ a different particle j at position $\vec{r}_j$ is considered a neighbour if $|\vec{r}_i - \vec{r}_j| < r_c$. If the force and potential energy of each particle is computed naively for each particle by considering the $(N-1)$ remaining atoms in the system then the resulting algorithm exhibits a computational complexity of $\mathcal{O}(N^2)$. However, in physically

realistic simulations of bulk material the spatial distribution of the N particles is typically approximately uniform, which implies that the number of neighbours of each particle is on average a constant. In chapter 3.1 we discuss in detail modern algorithms that compute the short-range interactions with $\mathcal{O}(N)$ computational complexity by using cell based methods to efficiently discard pairs of particles for which $|\vec{r}_j - \vec{r}_i| \gg r_c$.

In some areas of research, such as the study of Nematic Liquid Crystals (NLC), particles cannot be considered as point-wise objects as the potential energy between a pair of particles is a function of their relative orientations. Furthermore, the particular orientation of the particles can govern the observable property of interest, for example, the light polarisation of a system of NLC is determined by the collective orientation of the particles.

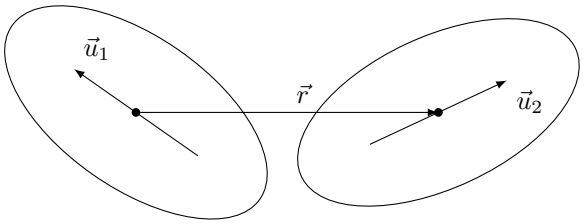

Figure 1-2: *Two ellipsoidal particles with axes of symmetry $\vec{u}_1$ and $\vec{u}_2$ separated by $\vec{r}$.*

The Gay-Berne [27] potential and derivatives are an anisotropic variant of Lennard-Jones interactions that can be applied between ellipsoidal shaped particles. The shape of each particle is determined by

$$G(x, y, x) = \exp\left(\frac{-(x^2 + y^2)}{\sigma_1^2} - \frac{z^2}{\sigma_2^2}\right), \tag{1.18}$$

where σ_1 and σ_2 are parameters and (x, y, z) is a point in a coordinate system local to the particle. The origin of this local coordinate system is at the centre of the particle and the z-axis is the axis of rotational symmetry. This description of particle shape does not describe an explicit boundary of the particle and the potential itself is formed by considering the overlap integral between two particles. If $\vec{u}_1$ and $\vec{u}_2$ specify the symmetrical axes of two ellipsoidal particles which centres are separated by the vector $\vec{r}$, as in Figure 1-2, then the

potential energy between the two particles is given by

$$U(\vec{u}_1, \vec{u}_2, \vec{r}) = \epsilon(\vec{u}_1, \vec{u}_2, \vec{r}) \left[\left(\frac{1}{1 + r - \sigma(\vec{u}_1, \vec{u}_2, \vec{r})} \right)^{12} - \left(\frac{1}{1 + r - \sigma(\vec{u}_1, \vec{u}_2, \vec{r})} \right)^{6} \right], \tag{1.19}$$

where
$$\epsilon(\vec{u}_1, \vec{u}_2, \vec{r}) = \bar{\epsilon}(\vec{u}_1, \vec{u}_2)^{\nu} \epsilon'(\vec{u}_1, \vec{u}_2, \vec{r})^{\mu}, \tag{1.20}$$

$$\bar{\epsilon}(\vec{u}_1, \vec{u}_2) = \epsilon_0 \left(1 - (\vec{u}_1 \cdot \vec{u}_2)^2 \chi^2\right)^{-1/2}, \tag{1.21}$$

$$\chi = \frac{\sigma_2^2 - \sigma_1^2}{\sigma_2^2 + \sigma_1^2}, \tag{1.22}$$

$$\epsilon'(\vec{u}_1, \vec{u}_2, \vec{r}) = 1 - \frac{\chi'}{2} \left(\frac{(\vec{r} \cdot \vec{u}_1 + \vec{r} \cdot \vec{u}_2)^2}{1 + (\vec{u}_1 \cdot \vec{u}_2)\chi'} + \frac{(\vec{r} \cdot \vec{u}_1 - \vec{r} \cdot \vec{u}_2)^2}{1 - (\vec{u}_1 \cdot \vec{u}_2)\chi'} \right), \tag{1.23}$$

$$\chi' = \frac{\epsilon_s^{1/\mu} - \epsilon_e^{1/\mu}}{\epsilon_s^{1/\mu} + \epsilon_e^{1/\mu}}, \tag{1.24}$$

$$\sigma(\vec{u}_1, \vec{u}_2, \vec{r}) = \sigma_0 \left[1 - \frac{\chi}{2} \left(\frac{(\vec{r} \cdot \vec{u}_1 + \vec{r} \cdot \vec{u}_2)^2}{1 + (\vec{u}_1 \cdot \vec{u}_2)\chi} + \frac{(\vec{r} \cdot \vec{u}_1 - \vec{r} \cdot \vec{u}_2)^2}{1 - (\vec{u}_1 \cdot \vec{u}_2)\chi} \right) \right]^{-1/2} \tag{1.25}$$

and ϵ_0, σ_0, ϵ_s, ϵ_e, μ and ν are constant parameters. In particular ϵ_s governes the side-to-side strength of the potential and ϵ_e governes the end-to-end strength. The Gay-Berne potential is mathematically and computationally more complex than the Lennard-Jones potential and requires a simulation code to store and make available the orientation of the particles.

In general, the potential between a pair of particles i and j is given by a function of the form

$$U_{ij} = U_{ij}(\pi^{(i)}, \pi^{(j)}), \tag{1.26}$$

where $\pi^{(i)}$ and $\pi^{(j)}$ are all properties of the particles, for example, position and charge in the Lennard-Jones example or shape parameters in the Gay-Berne example. This two-body potential can be extended to a general n-body potential between particles $(1, \ldots, n)$ as

$$U_{1,\ldots,n} = U_{1,\ldots,n}(\pi^{(1)}, \ldots, \pi^{(n)}), \tag{1.27}$$

where $\{\pi^{(1)}, \ldots, \pi^{(n)}\}$ are the particle properties. In Section 2.2 we present an abstraction for pair-wise operations between particles in which two-body potentials can easily be described.

Time Integration: An Energy Conserving Scheme

Given an inter-particle potential, such as a Lennard-Jones potential, and a set of N particle positions $\{\vec{r}_i, i = 1 \ldots N\}$ the force $\vec{F}_i$ exerted on each particle i and the system potential energy $\mathcal{U}$ can be evaluated from Equations (1.16, 1.17). Hence from Equation (1.10) (Newton's Second Law) the acceleration of each particle can be computed and used to update the velocities and subsequently the positions of all particles via a time integration scheme. If the simulation is conducted in an ensemble where the total energy should remain

constant, e.g. the micro-canonical ensemble, then the time integration scheme applied to the system must be energy conserving.

We use the Velocity Verlet [79, 77] scheme to integrate the system in time as this time stepping scheme is energy conserving, locally 4^{th} order accurate and globally 2^{nd} order accurate. The Velocity Verlet scheme is a leapfrog method and is given in Algorithm 1, where we use superscript n to denote the n^{th} iteration.

Algorithm 1: Time evolution using Velocity Verlet with time step δt. In this example we use an inter-particle force that is a function of particle positions, in general this force can be a function of all particle properties.

Data: Previous positions $\{\vec{r_i}^{\,n}, i = 1 \dots N\}$, velocities $\{\vec{v_i}^{\,n}, i = 1 \dots N\}$ and accelerations $\{\vec{a_i}^{\,n}, i = 1 \dots N\}$.

Result: New positions $\{\vec{r_i}^{\,n+1}, i = 1 \dots N\}$, velocities $\{\vec{v_i}^{\,n+1}, i = 1 \dots N\}$ and accelerations $\{\vec{a_i}^{\,n+1}, i = 1 \dots N\}$.

Compute half update of velocities:	$\vec{v_i}^{\,n+\frac{1}{2}} = \vec{v_i}^{\,n} + \frac{1}{2}\,\vec{a_i}^{\,n}\,\delta t$
Compute full update of positions:	$\vec{r_i}^{\,n+1} = \vec{r_i}^{\,n} + \vec{v_i}^{\,n+\frac{1}{2}}\,\delta t$
Compute new accelerations:	$\vec{a_i}^{\,n+1} = \frac{1}{m_i}\vec{F_i}(\vec{r_1}^{\,n+1}, \dots, \vec{r_N}^{\,n+1})$
Compute remaining half update of velocities:	$\vec{v_i}^{\,n+1} = \vec{v_i}^{\,n+\frac{1}{2}} + \frac{1}{2}\,\vec{a_i}^{\,n+1}\,\delta t$

Computationally the most expensive component is the calculation of the new forces $\vec{F_i}$ required by the third step in Algorithm 1. Similarly, within a MC simulation the bottleneck is the computation of the new potential energy. Given a method to compute the particle forces, we may integrate the system forward in time to construct a trajectory. Each instance of positions and velocities produced by the evolution of the system of particles is a point in the phase space of the system.

Thermostats

In the Canonical ensemble the simulated system is considered to be thermodynamically coupled to a heat bath of constant temperature such that the system and the heat bath are in thermal equilibrium. This heat bath is considered to have a large enough thermal mass that fluctuations in the temperature of the simulated system do not change the temperature of the bath.

The Andersen thermostat [8] is a numerical method to approximately couple the simulated system to the heat bath via a mechanism where particles stochastically collide with the heat bath. When a particle collides with the heat bath it is assigned a new velocity which is sampled from the Maxwell-Boltzmann distribution [50, 49, 11, 12] that corresponds to the temperature T of the heat bath. The Maxwell-Boltzmann distribution is the probability distribution of particle velocities found within a system at a given

temperature.

The strength of the coupling between the particles and the heat bath is governed by a parameter ν which determines the frequency of collisions. For a given ν the distribution of time intervals between collisions is a Poisson distribution with parameter ν. Hence the probability of zero collisions in a time interval t is

$$P(t) = \nu \exp(-\nu t). \tag{1.28}$$

When a particle collides with the heat bath a new velocity is drawn from the Maxwell-Boltzmann distribution, to sample from this distribution each component of the new velocity is individually sampled from a Gaussian distribution with zero mean and variance $\sqrt{T}$. As the Andersen thermostat samples an entirely new velocity vector when a particle collides with the heat bath the method generates velocities which are discontinuous, these discontinuities indicate that the particle dynamics are not entirely representative of a physical system. If the study of a system requires highly representative particle dynamics then there exist extended Lagrangian thermostats such as Nosé-Hoover [55, 54, 38, 39].

Hybrid Monte Carlo

Hybrid Monte Carlo [19, 52] (HMC), also known as Hamiltonian Monte Carlo, is an importance sampling technique that combines the ideas of MC with Hamiltonian dynamics. Consider a system where the phase space is partitioned into multiple regions separated by energy barriers. It is expected that the trajectory a MD simulation follows in the micro-canonical ensemble will become trapped in a local energy minima and hence not be ergodic. A MC approach will accept proposed states based on the Boltzmann distribution, as this distribution is highly peaked large proposed steps will have an unacceptably low acceptance rate, small proposed steps have a higher acceptance rate but become highly correlated. HMC allows larger steps through phase space to be proposed without small acceptance rates and with much smaller correlations between samples.

Hamiltonian dynamics describe the evolution of a point through a phase space of $2d$ dimensions. Each point in the Hamiltonian phase space corresponds to the combination of a d element vector $\vec{Q}$ of Hamiltonian positions and a d element vector $\vec{P}$ of Hamiltonian momenta. Consider a system of N particles with positions $\vec{r} \in \mathbb{R}^{3N}$ and momenta $\vec{p} \in \mathbb{R}^{3N}$. Let $\vec{Q} = (\vec{r}, \vec{p}) = (Q_{\vec{r}}, Q_{\vec{p}}) \in \mathbb{R}^{6N}$ be the Hamiltonian position that corresponds to this point in phase space and let $\vec{P} \in \mathbb{R}^{6N}$ be the momenta of the Hamiltonian positions $\vec{Q}$.

Given $\vec{Q}$ and $\vec{P}$ the time evolution though the Hamiltonian phase space is governed by

Hamilton's equations

$$\frac{d\vec{Q}_i}{dt} = \frac{\partial H\left(\vec{Q},\vec{P}\right)}{\partial \vec{P}_i}, \tag{1.29}$$

$$\frac{d\vec{P}_i}{dt} = -\frac{\partial H\left(\vec{Q},\vec{P}\right)}{\partial \vec{Q}_i}, \tag{1.30}$$

where $H\left(Q,P\right)$ is the Hamiltonian of the system. For HMC the Hamiltonian is constructed as

$$H\left(\vec{Q},\vec{P}\right) = \mathcal{U}^{(\mathrm{H})}(\vec{Q}) + \mathcal{K}^{(\mathrm{H})}(\vec{P}), \tag{1.31}$$

where $\mathcal{U}^{(\mathrm{H})}$ is referred to as the Hamiltonian potential energy and $\mathcal{K}^{(\mathrm{H})}$ is the Hamiltonian kinetic energy. The Hamiltonian potential energy is given by

$$\mathcal{U}^{(\mathrm{H})} = -\log\left(\rho\left(\vec{Q}\right)\right), \tag{1.32}$$

where $\rho\left(\vec{Q}\right)$ is the probability density distribution to sample from. We wish to sample from the Boltzmann distribution:

$$\rho\left(\vec{Q}\right) = \frac{1}{Z}\exp\left(-\frac{\mathcal{U}(\vec{Q}_{\vec{r}}) + \mathcal{K}(\vec{Q}_{\vec{p}})}{k_B T}\right), \tag{1.33}$$

where $\vec{Q}_{\vec{r}} = \vec{r}$ are the Hamiltonian positions representing particle positions and $\vec{Q}_{\vec{p}} = \vec{p}$ are the Hamiltonian positions representing particle momenta, $\mathcal{U}$ is the inter-particle potential energy, $\mathcal{K}$ is the total kinetic energy of the particle system and Z is the normalisation constant. Hence the Hamiltonian potential energy can be written more explicitly as

$$\mathcal{U}^{(\mathrm{H})} = \frac{\mathcal{U}(\vec{r}) + \mathcal{K}(\vec{p})}{k_B T} - \log(Z). \tag{1.34}$$

The Hamiltonian kinetic energy $\mathcal{K}^{(\mathrm{H})}$ is defined as the logarithm of the probability density of the momenta distribution

$$\mathcal{K}^{(\mathrm{H})}\left(\vec{P}\right) = \frac{1}{2}\vec{P}^{T}M^{-1}\vec{P}, \tag{1.35}$$

where M is the covariance matrix of the multivariate Gaussian which we assume is diagonal and the Hamiltonian momenta $\vec{P}$ are samples from a multivariate Gaussian distribution with zero mean.

By using a symplectic integrator Hamilton's equations can be integrated from some initial state at t_0 to a time t_1 whilst (approximately) conserving the value of the Hamiltonian.

A leapfrog scheme for Hamilton's equations with time-step size δt is given by

$$\vec{P}_i^{\,t+\frac{\delta t}{2}} = \vec{P}_i^{\,t} + \frac{\delta t}{2}\frac{\partial U^{(\mathrm{H})}}{\partial \vec{Q}_i}\left(\vec{Q}^t\right), \tag{1.36}$$

$$\vec{Q}_i^{\,t+\delta t} = \vec{Q}_i^{\,t} + \frac{\delta t}{m_i}\vec{P}_i^{\,t+\frac{\delta t}{2}}, \tag{1.37}$$

$$\vec{P}_i^{\,t+\delta t} = \vec{P}_i^{\,t+\frac{\delta t}{2}} + \frac{\delta t}{2}\frac{\partial U^{(\mathrm{H})}}{\partial \vec{Q}_i}\left(\vec{Q}^{t+\delta t}\right), \tag{1.38}$$

where m_i is the i^{th} diagonal entry of M, for $\vec{P}$ and $\vec{Q}$ subscript i denotes i^{th} component and superscript t denotes the t^{th} time step. This integration stage produces a sample from the probability distribution $\rho(\vec{Q}, \vec{P})$ where

$$\rho\left(\vec{Q}, \vec{P}\right) = \exp\left(-H\left(\vec{Q}, \vec{P}\right)\right), \tag{1.39}$$

$$= \rho\left(\vec{Q}\right)\rho\left(\vec{P}\right). \tag{1.40}$$

The generated distribution of $\vec{Q}$ from Hamilton's equations is given by the marginal distribution

$$\int \rho\left(\vec{Q}\right)\rho\left(\vec{P}\right)d\vec{P} = \rho\left(\vec{Q}\right). \tag{1.41}$$

Hence by constructing the Hamiltonian H in this way the trajectory through the Hamiltonian phase space generates samples from the original distribution that we wish to sample from. Furthermore, if we could numerically integrate Hamilton's equations forward in time exactly then the resultant scheme would be rejection free. Using the preceding machinery an iteration of the HMC algorithm is a two step process:

1. The existing configuration of particle positions and momenta are collectively considered as the initial ($t = t_0$) Hamiltonian positions $\vec{Q}$ and initial Hamiltonian momenta $\vec{P}$ are sampled from a multivariate normal distribution. Using a symplectic integrator Hamilton's equations are integrated forward to some end time ($t = t_1$).

2. The resulting system at $t = t_1$ is considered as a proposed configuration and is accepted with probability

$$\begin{aligned}&\min\left[\exp\left(H\left(\vec{Q}^{\,t_0}, \vec{P}^{\,t_0}\right) - H\left(\vec{Q}^{\,t_1}, \vec{P}^{\,t_1}\right)\right), 1\right] = \\ &\min\left[\exp\left(\mathcal{U}^{(\mathrm{H})}(\vec{Q}^{\,t_0}) + \mathcal{K}^{(\mathrm{H})}(\vec{P}^{\,t_0}) - \mathcal{U}^{(\mathrm{H})}(\vec{Q}^{\,t_1}) - \mathcal{K}^{(\mathrm{H})}(\vec{P}^{\,t_1})\right), 1\right].\end{aligned} \tag{1.42}$$

   If the proposed state is rejected then the original state is reused as the next state. This step exists to correct for numerical error in the time integration scheme.

As discussed by Neal [52] the Hamiltonian momenta must be sampled from the Gaussian distribution at each iteration of the algorithm to ensure that the Hamiltonian positions are samples from the desired distribution.

Identifying Common Operations

We identified that in general a n-body potential can be written as $U_{1,\ldots,n}(\pi^{(1)},\ldots,\pi^{(n)})$ where $\{\pi^{(1)},\ldots,\pi^{(n)}\}$ are particle properties. In addition to inter-particle interactions, the MD algorithms we have described all read and modify particle properties in a uniform manner, i.e. a function G is applied such that $\pi^{(i)} \leftarrow G(\pi^{(i)})$ for all particles i. We now describe a selection of existing analysis algorithms, all of which can be described in terms of n-body operations and operations that uniformly apply a function to the properties of each particle.

1.1.4 Analysis Techniques

A simulation of a system of particles is of little use without a method to extract useful information. As we introduced in the statistical mechanics section, an observable of the system is some function of the state of the simulation. The most simple observables are the potential and kinetic energy which are scalar valued quantities that are routinely computed within simulations at either all time-steps or periodically. We present a limited set of existing analysis techniques with varying algorithmic and computational complexities that are representative of how typical analysis algorithms operate.

Diffusion Coefficients

Often researchers are interested in the study of complex processes that occur in the simulated system and these processes could be time dependent. For example, the Mean Squared Displacement (MSD) is a technique that can be used to estimate the diffusion coefficients in a system. For example, a researcher could investigate the average motion of particles with type a in a system predominately filled with particles of type b by measuring the diffusion coefficient. Fick's second law states that the rate of change of concentration of a diffusing quantity c is proportional to the divergence of the flux created by the gradient of c,

$$\frac{\partial c}{\partial t} = \vec{\nabla} \cdot \left(D \vec{\nabla} c \right), \tag{1.43}$$

where D is the diffusion coefficient that we wish to measure. Einstein is attributed with the relation

$$\frac{\partial \langle \vec{r}^2 \rangle}{\partial t} = 6D, \tag{1.44}$$

where $\langle \vec{r}^2 \rangle$ is the average MSD. Over a sufficiently long time frame the value of D can be estimated from a simulation by the relation

$$\langle \vec{r}^2 \rangle = \lim_{r \to \infty} 6tD. \tag{1.45}$$

To compute an estimate of D in practice requires that a simulation code records the positions of all particles at multiple times to compute the value of the MSD. For systems at equilibrium, diffusion coefficients can also be computed by using the velocity autocorrela-

tion function (VACF) which measures the correlation between the velocity of a particle at an initial time t_0 and a future time t. The VACF of a particle i between an initial time t_0 and a time of measurement t is defined as $\vec{v}_i(t_0) \cdot \vec{v}_i(t)$ where $\vec{v}_i(t)$ is the velocity of the particle at time t. When evaluated on a system of particles at equilibrium the VACF is invariant under a change of initial time t_0 such that

$$\langle \vec{v}(t_0) \cdot \vec{v}(t) \rangle = \langle \vec{v}(0) \cdot \vec{v}(t - t_0) \rangle. \tag{1.46}$$

Given a method to compute the VACF in a simulation, the diffusion coefficient D can be subsequently estimated by the relation

$$D = \frac{1}{3} \int_0^\infty \langle \vec{v}(0) \cdot \vec{v}(\tau) \rangle d\tau. \tag{1.47}$$

A generalisation of Equation (1.47) for computing transport coefficients in terms of the integrals of time correlation functions is given by the Green-Kubo relations [29, 46].

A VACF calculation could be computed after the simulation by using a trajectory dump where the velocity of each particle is recorded at a series of time-steps. Implementation of an on-the-fly VACF calculation that is computed within the simulation itself requires the velocity vectors $\vec{v}_i(0)$ and $\vec{v}_i(t)$ for each particle i. Hence the simulation software should provide data structures to store an initial set of velocities $\vec{v}_i(0)$ and a current set of velocities $\vec{v}_i(t)$.

Given data structures to store initial and current velocities the VACF can be computed by looping over all particles and incrementing a variable that stores a running total of the VACF with the contribution from each particle. Initial velocities can be reset by looping over all particles and copying the value of the current velocity $\vec{v}_i(t)$ into the data structure that stores $\vec{v}_i(0)$.

Structure Analysis

An area of interest that requires more advanced analysis techniques is the study of the local environments of particles within a simulation [48]. For particles in a crystal structure there exist methods to classify the crystal structure type a particular particle is a member of. This classification is necessary as the crystalline structure formed by cooling a liquid of identical particles is not unique. Consider a liquid system of identical particles that interact with a Lennard-Jones potential which includes an attractive and repulsive component. As the system is cooled by removing kinetic energy through a thermostat, such as an Andersen thermostat, the liquid system will pass through the liquid to solid phase transition and form a crystalline structure. The crystalline structure formed corresponds to an arrangement of particles that attempts to minimise the potential energy. As the minimum of the Lennard-Jones potential is at an inter-particle separation of $r = 2^{1/6}\sigma$, from a macroscopic viewpoint, the system is expected to try and maximise the number of inter-particle distances around $r \approx 2^{1/6}\sigma$, producing a crystal structure.

In 3D there exist two common lattices which satisfy the separation requirement, namely face-centred cubic (fcc) and hexagonal close-packed (hcp). Furthermore, as the system of particles forms a solid it is highly likely that the solid will be a mixture of both fcc and hcp and non-classified lattice types. There also exist icosahedral structures that locally minimise the energy between a subset of particles but cannot be the basis of a lattice. In the study of glasses [70] and self assembly distinguishing between these structures is an active area of research.

The bond order analysis (BOA) technique [75] introduces a set of order parameters which are defined for each particle i as

$$Q_\ell^{(i)} = \sqrt{\frac{4\pi}{2\ell+1}\sum_{m=-\ell}^{+\ell}|q_{\ell m}^{(i)}|^2}, \tag{1.48}$$

with $\ell = 0, 1, 2, \ldots$. The sum

$$q_{\ell m}^{(i)} = \frac{1}{|\mathcal{N}(i)|}\sum_{j\in\mathcal{N}(i)} Y_\ell^m(\hat{\vec{r}}_{ij}), \tag{1.49}$$

is computed by evaluating the spherical harmonics Y_ℓ^m in the directions

$$\hat{\vec{r}}_{ij} = \frac{\vec{r}_i - \vec{r}_j}{|\vec{r}_i - \vec{r}_j|},$$

pointing from the atom i to each of its neighbours $j \in \mathcal{N}(i)$. The BOA method uses the following definition of the spherical harmonics,

$$Y_l^m(\theta,\phi) = \sqrt{\frac{2l+1}{4\pi}\frac{(l-m)!}{(l+m)!}}e^{im\phi}P_l^m\left(\cos\left(\theta\right)\right), \tag{1.50}$$

where P_l^m denotes the Associated Legendre Polynomial of order (l, m), ϕ is the azimuthal angle of $\hat{\vec{r}}_{ij}$ and θ is the polar angle of $\hat{\vec{r}}_{ij}$. Atoms are considered to be neighbours if their distance is smaller than a predefined cutoff range r_c. Perfect crystal lattices have well defined values for Q_ℓ. In particular the order parameters with $\ell = 4, 5, 6$ are often used to estimate the degree and type of crystal. Specific values for fcc, hcp and bcc lattices are given in Table 1.1 ([76, 51]).

Lattice Structure	Q_4	Q_5	Q_6
fcc	0.191	0	0.575
hcp	0.097	0.252	0.485
bcc	0.036	0	0.511

Table 1.1: *Values of Q_4, Q_5 and Q_6 for perfect lattices, see [76] and Table 1 in [51].*

In a simulation the local structure of the material can therefore be estimated by calcu-

lating $Q_\ell^{(i)}$ and comparing to the reference values in Table1.1. If they agree within some tolerance, the system is classified to be in the corresponding lattice.

The second local environment analysis method we consider is common neighbour analysis (CNA) [37]. CNA is a purely topological approach to classify the local environment of each particle by considering bonded particles. A pair of particles are considered to be bonded if they exist within a cutoff distance r_c of each other. For a pair of bonded particles (i, j) the set of all particles which are bonded to both i and j are referred to as *common neighbours*, the bonds between the common neighbours define a graph $\mathcal{G}$. For each pair of bonded particles (i, j) the graph of common neighbours $\mathcal{G}$ is classified by three numbers [76]: (1) the number common neighbours n_{nb} i.e. the number of vertices in $\mathcal{G}$, (2) the number of bonds n_{b} i.e. the number of edges in $\mathcal{G}$, and (3) n_{lcb} the number of bonds in the largest connected subgraph $\mathcal{G}' \subset \mathcal{G}$. For each pair of bonded particles these define a triplet $(n_{\text{nb}}, n_{\text{b}}, n_{\text{lcb}})$ (example in Figure 1-3).

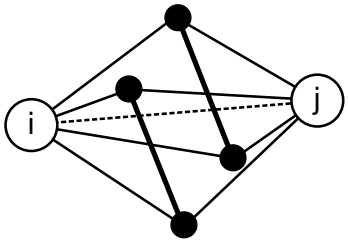

Figure 1-3: *Common neighbour analysis for bonded atom pair (i, j) (empty circles). The set of common neighbours (filled circles) are classified as a $(4, 2, 1)$ triplet.*

To classify the local environment of a particle the triplets are computed for each bonded neighbour and compared to reference signatures for crystal structures. For example, in a hcp lattice, each atom has 12 bonds, six of which are classified as $(4, 2, 1)$ and the other six are $(4, 2, 2)$; see Table 1 in [76].

We consider the calculation of the BOA for a set of particles as a two stage process. In the first stage for each particle the neighbouring particles which are sufficiently close to be considered as bonded are counted, hence each particle requires data storage for the integer number of neighbouring particles. For each particle the second stage loops over all nearby particles and computes the BOA parameters for each neighbour and increments the values of Q_4, Q_5, Q_6 on each particle, hence storage should be provided for three floating point values per particle.

Analysis Conclusions

The implementation of more involved analysis methods, such as BOA and CNA, presents a technical challenge as the methods are often computationally expensive. Most simulation software provides the capability to periodically store snapshots of the simulation which can be used as inputs to a stand-alone tool that performs the analysis. Individually a

snapshot can require a non-trivial quantity of storage and hence the storage required to store a trajectory for post-processing analysis can be significant.

Analysis could be performed within the simulation on-the-fly, this avoids the cost of storing and reading data which is a relatively slow process. However, to perform on-the-fly processing the analysis algorithm must be implemented within the simulation software which requires detailed knowledge of the inner workings of the code. As we have outlined, performing on-the-fly analysis requires simulation software to allow additional data structures for particle properties and provide access to fundamental loops over particles and their neighbours. Typically, this code modification direction requires technical knowledge outside the area of expertise of the domain specialist performing the simulation and analysis. However, the analysis methods we have described can be described by loops over particles and loops over pairs of particles that manipulate a general set of particle data.

Often computationally expensive analysis methods are implemented as sequential standalone programs which do not exploit available parallelism, typically due to a lack of the technical expertise required to implement such algorithms in existing low-level programming languages. The data structures and looping mechanisms that are required to produce a MD simulation framework are the same as those required for many analysis methods. A framework that allows a extensible method to produce parallel MD simulations can also provide a flexible and parallel analysis environment for on-the-fly or standalone analysis.

1.2 Modern High Performance Computing

In scientific computing researchers develop and implement algorithms that compute solutions to large scale problems such as weather prediction and material simulation. Typically, the initial implementations of algorithms are created as prototype software for commodity hardware, such as laptop and desktop computers, and these implementations demonstrate that the underlying algorithm is a viable method. Often this commodity hardware provides insufficient computational performance to apply the method to the original large scale problem and a High Performance Computing (HPC) facility is required. For example, it would be possible in theory to compute a weather forecast on a laptop, however it would take an extremely long time for an accurate result which would then be useless as a forecast as the weather would have already occurred.

The weather prediction example is a case where a HPC facility is required to produce any reasonable result at all. We focus on MD simulation for material modelling where HPC facilities enable the study of larger systems and drastically reduce the calculation time required to perform simulations in comparison to commodity hardware. For instance, a large MD simulation could require weeks or months of compute time on a desktop computer but by using a HPC facility results could be computed in days.

As a further example, consider an experiment where the same simulation needs to be performed for a large number of parameters and each simulation performed is computationally non-trivial. The simulation for each parameter could be computed in turn on a

single desktop computer, which may take an unreasonable amount of time, but by using a HPC facility the overall time to solution is greatly reduced enabling a higher throughput of results.

1.2.1 General HPC Facility Topology

Most modern HPC facilities share a generic overall structure to provide scalable performance. This structure consists of a collection of compute nodes that perform the computations and are interconnected via a high performance network. A HPC facility also typically contains a large volume of storage for data required by computations and data produced by computations and an interface for users to access the computer through. In this section we cover the general architecture of the computational portion of a HPC facility and the requirements software developers must meet to efficiently use the resource.

A modern compute node is structurally similar to a high performance desktop workstation and often consists of multiple Central Processing Units (CPUs), a large volume of fast but volatile system memory commonly known as Random Access Memory (RAM) and a fast network interface to connect to other resources in the facility. The network interface is used to connect compute nodes together via specialist interconnects, such as those based on the InfiniBand standard, or proprietary technology such as Intel Omni-Path. The rate at which data is transferred across the network is referred to as the bandwidth and the minimum time required to transfer any data is referred to as the latency. These high speed networks offer point-to-point bandwidths in the region of 100 Gb/s which is much higher than conventional Ethernet that provides a bandwidth in the region of 10 Gb/s. Following convention, we list network bandwidths in units of Gigabits per second (Gb/s) as opposed to Gigabytes per second (GB/s). Furthermore, specialist networks provide a point-to-point latency ≈ 1 μs which is lower than conventional Ethernet. This is important as for inter-node communication it is common that the latency of the network has a higher impact on efficiency than the bandwidth available.

1.2.2 Compute Nodes

Memory

Compute nodes contain a hierarchy of volatile memory technologies that store in-use values, the term volatile is used to indicate that these memory types require continuous power to retain data. The largest pool of volatile memory with a volume of 10 GB - 2 TB within a compute node is the main system memory which is connected directly to the CPUs with a hardware link capable of ≈200 GB/s on a modern system. CPUs themselves contain multiple banks of memory known as the CPU cache with capacities that vary in the region of 256 kB - 100 MB and bandwidths that vary in the region of 3 TB/s - 100 GB/s, as a general rule of thumb higher bandwidth cache is significantly more expensive than lower bandwidth cache. Hence CPUs contain a hierarchy of cache levels that vary from a small capacity cache per core with a high bandwidth and low latency

to the largest cache with the lowest bandwidth and highest latency, this larger cache may be shared across multiple cores. Values which are repeatedly required by a program are stored locally within a CPU cache and are accessed much more quickly than if the value was re-fetched from main system memory.

CPUs

Along with banks of CPU cache and memory controllers a CPU contains multiple identical units referred to as "cores" that perform operations on input data. The CPU operations that are the most relevant in many scientific codes are those that operate on floating point values. A floating point value is an approximation to a real valued number that is made with 32bits (single precision) or 64bits (double precision). A common unit used to measure rates of computation is the Giga-Floating Point Operation per Second (GFLOPs) which we will exclusively use to refer to operations performed on double precision values. The term "operation" refers to the mathematical operation that the core is performing, for example, consider the task $d \leftarrow ab + c$ which is read as "d is assigned the value of a times b plus c". This simple example contains two mathematical operations (addition and multiplication) and four memory references. We make the distinction between mathematical operations and CPU operations as they are not equivalent. The two mathematical operations in this example are performed by one operation in certain cores by an instruction known as a Fused Multiply Add (FMA).

A popular compute node configuration at the time of writing consists of two CPUs within a node, each CPU will be comprised of a number of cores as shown in Figure 1-4. The exact number of compute cores per CPU varies between vendors and individual models, at the time of writing modern CPU core counts are in the range 2-72.

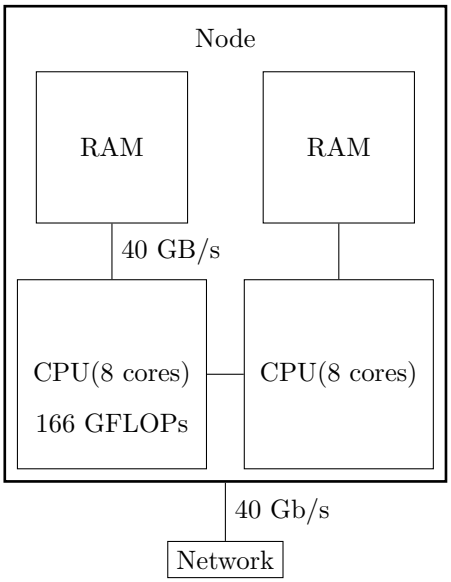

Figure 1-4: *Example node configuration comprised of two CPUs each with 8 compute cores. Performance numbers for memory bandwidth, compute rate and network bandwidth are representative of an Intel E5-2650v2 (2.6 GHz Ivy Bridge) combined with an Intel TrueScale QDR network interface.*

Accelerators

CPUs are capable of executing very general sets of instructions quickly whilst being relatively easy to program. This flexibility alongside a history of relatively cheap prices has made them the most ubiquitous computing resource in HPC. Advances in other computing architectures such as Graphics Processing Units (GPUs), Field Programmable Gate Arrays (FPGAs) and Application-Specific Integrated Circuits (ASICs) have provided alternative approaches to perform computations. In comparison to CPUs these accelerators may provide higher memory bandwidth and higher computation rates but only if the implemented algorithm is suitable for the hardware.

These alternative architectures can be included in node configurations alongside traditional CPUs and communicate with the CPU and system memory through a high speed connection such as Peripheral Component Interconnect Express (PCIe). If an accelerator device operates on memory built into the accelerator itself then any required data must be transferred from main system to the accelerator over the interconnect and results copied back in the reverse direction. The time taken to transfer data to and from the accelerator may negate any performance benefit gained by offloading computation onto the accelerator, and hence successful application of accelerators requires careful software design. An example node configuration where a pair of CPUs have been combined with a pair of accelerators is given in Figure 1-5.

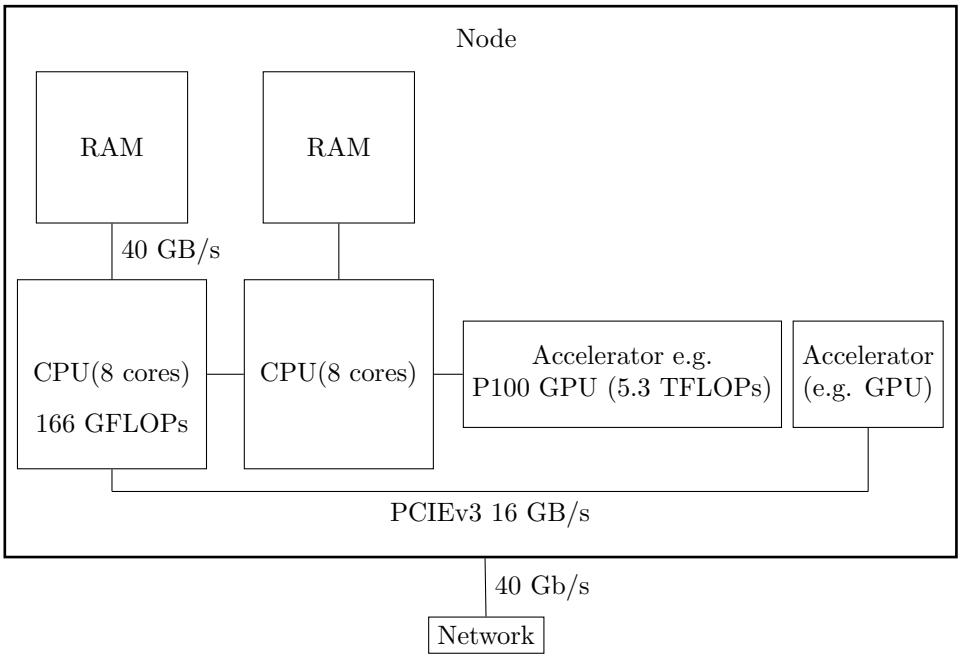

Figure 1-5: *Example node configuration comprised of two CPUs each with 8 compute cores paired with some accelerator. Performance numbers are representative of an Intel E5-2650v2 (2.6 GHz Ivy Bridge) system combined with an Intel TrueScale network interface.*

Floating Point Units (FPU)

In general, the performance of a MD simulation implementation is bound by the achieved floating point computation rate of on the HPC hardware as opposed to the rate at which values are retrieved from memory. We now describe how modern CPUs perform floating point operations as this background material is required knowledge for implementing and analysing the efficiency of CPU code.

Due to physical constraints regarding heat dissipation and power delivery CPU core clock speeds have remained reasonably static within the past two decades (2-4 GHz). CPU manufacturers have increased computational performance by increasing the number of cores per CPU and increasing the number of instructions that can be executed per clock cycle per core. To achieve more operations performed per clock cycle CPU cores have silicon dedicated to specialised tasks, such as vector operations and to manage the flow of instructions to maximise throughput. To efficiently use a modern CPU for purely floating point computation a program must use these vector instructions. Vector operations are a form of Single Instruction Multiple Data (SIMD) parallelism, where the same mathematical operations are applied to multiple independent sets of data in parallel.

The width of the vector FPU determines how many elements are processed per clock cycle. In modern CPUs the vector width can vary from 128bit (2 double precision values) to 512bit (8 double precision values). As an example consider the Intel E5-2650v2 processor with eight cores and a clock-speed of 2.6 GHz, each core of this CPU contains a vector addition unit and a vector multiplication unit both with a width of 256bit. These vector

units operate on input vectors concurrently and can perform one vector addition and multiplication per clock cycle assuming the input data is available. If we define $\mathbb{R}_{64}$ to be the set of real numbers representable in double precision floating point arithmetic the multiplication unit and addition unit perform the operations defined by vadd and vmul as element-wise addition and multiplication:

$$\text{vadd}\colon \mathbb{R}_{64}^4 \times \mathbb{R}_{64}^4 \mapsto \mathbb{R}_{64}^4, \tag{1.51}$$

$$\text{vadd}\left(\begin{bmatrix} a_0 \\ a_1 \\ a_2 \\ a_3 \end{bmatrix}, \begin{bmatrix} b_0 \\ b_1 \\ b_2 \\ b_3 \end{bmatrix}\right) \mapsto \begin{bmatrix} a_0 + b_0 \\ a_1 + b_1 \\ a_2 + b_2 \\ a_3 + b_3 \end{bmatrix}, \tag{1.52}$$

$$\text{vmul}\colon \mathbb{R}_{64}^4 \times \mathbb{R}_{64}^4 \mapsto \mathbb{R}_{64}^4, \tag{1.53}$$

$$\text{vmul}\left(\begin{bmatrix} a_0 \\ a_1 \\ a_2 \\ a_3 \end{bmatrix}, \begin{bmatrix} b_0 \\ b_1 \\ b_2 \\ b_3 \end{bmatrix}\right) \mapsto \begin{bmatrix} a_0 b_0 \\ a_1 b_1 \\ a_2 b_2 \\ a_3 b_3 \end{bmatrix}. \tag{1.54}$$

Certain CPU architectures, such as the Intel Haswell architecture, can execute the FMA instruction on vectors of double and single precision values in a similar element-wise fashion,

$$\text{vfma}\colon \mathbb{R}_{64}^4 \times \mathbb{R}_{64}^4 \times \mathbb{R}_{64}^4 \mapsto \mathbb{R}_{64}^4, \tag{1.55}$$

$$\text{vfma}\left(\begin{bmatrix} a_0 \\ a_1 \\ a_2 \\ a_3 \end{bmatrix}, \begin{bmatrix} b_0 \\ b_1 \\ b_2 \\ b_3 \end{bmatrix}, \begin{bmatrix} c_0 \\ c_1 \\ c_2 \\ c_3 \end{bmatrix}\right) \mapsto \begin{bmatrix} a_0 b_0 + c_0 \\ a_1 b_1 + c_1 \\ a_2 b_2 + c_2 \\ a_3 b_3 + c_3 \end{bmatrix}. \tag{1.56}$$

On modern CPUs the vector instructions are the most compute intensive instructions a CPU core executes, i.e. no other instruction performs more mathematical operations in the same or fewer clock cycles, hence the vector instructions are used to compute the theoretical peak computation rate. If we again consider the Intel E5-2650v2 where each core contains a separate 256bit vector addition and multiplication unit that can perform a vadd and vmul once per clock cycle then each CPU core could perform eight double precision mathematical operations per cycle (8 FLOPs). Hence at a fixed clock-speed of 2.6 GHz each core of a E5-2650v2 attains a maximum computation rate of 8 FLOP$*$2.6 GHz = 20.8 GFLOPs and the theoretical maximum computation rate of the eight cores is 166.4 GFLOPs.

Compared to the rate a modern CPU can perform computation fetching data from memory is comparatively slow. For example, the E5-2650v2 can perform floating point operations at a rate of 166.4 GFLOPs and has a memory bandwidth in the region of 40 GB/s which is 5×10^9 double precision values per second. The ratio between the system

memory bandwidth and the computation rate of the CPU indicates that an E5-2650v2 can theoretically perform ≈ 33 mathematical operations in the time taken to retrieve a single double precision value from memory. Hence the performance critical portion of a program must reuse values read from memory for a significant number of compute operations to achieve a significant proportion of the peak computation rate.

For a section of code the ratio between mathematical operations performed and required memory bandwidth to main memory is known as the arithmetic intensity and is commonly measured in FLOPs/Byte. Hence our E5-2650v2 requires an implementation to achieve an arithmetic intensity of at least 4.2 FLOPs/Byte to theoretically attain peak performance. Our peak performance calculation required both addition and multiplication units to be fully utilised on each clock cycle, hence to approach peak performance an implementation should contain an equal number of additions and multiplications and the data available to perform the computation. In practice codes contain different numbers of additions and multiplications and hence cannot approach peak performance.

Graphics Processing Units (GPUs)

GPUs can be viewed as a very wide vector processor which operates on thousands of elements simultaneously. Although initially designed to accelerate 2D and 3D graphics these devices are programmable for general purpose computation. From a high level viewpoint, GPU code must be designed in a SIMD manner such that many (thousands) of threads can apply the same operation to independent sets of data. GPUs typically offer higher peak performance than CPUs by containing thousands of cores per GPU and often operating with a higher power draw limit. GPUs are typically packaged as an accelerator card that connects to the rest of the host system via the PCIe interface and include a separate bank of memory that is directly connected to the GPU. The Tesla P100 [57] manufactured by Nvidia features 3584 cores with a power draw limit of 250W and achieves a peak double precision execution rate of 5.3 TFLOPs using FMA instructions, this floating point performance is paired with a peak memory bandwidth of 720GB/s.

In contrast to GPU cores CPU cores dedicate a large proportion of silicon space to functionality that aims to optimise the flow of instructions and data to the CPU execution units, for example, the floating point vector units. In modern GPU architectures, such as those manufactured by Nvidia, execution cores are grouped in sizes of 32 into units referred to as Streaming Multiprocessors (SMs) all 32 cores in a SM execute the same instructions simultaneously. The scheduling of instructions is performed on a per SM basis as all cores in the SM execute the same instructions. By grouping the cores into SMs the average area of silicon space required per core is greatly reduced.

Although a GPU core may only execute instructions from a single thread per clock cycle GPUs architectures are designed such that the cores host multiple execution threads simultaneously and incorporate thread scheduling that allows cores to switch execution between threads efficiently. The idea is that if the execution of a thread stalls whilst

waiting for data to be fetched from or stored into the GPU memory then the core can switch execution to a thread which has no outstanding data movement dependencies and can continue executing instructions.

The large peak memory bandwidth and computation rate of a GPU is a driving factor for developing codes for these accelerators, especially for computation bound code such as the force calculation in MD. The introduction of a GPU into a HPC node further complicates the memory hierarchy as the device operates on values stored in the onboard memory, which is connected to the rest of the node through the PCIe bus. A significant implementation challenge in programming GPUs is the task of structuring the algorithm such that the PCIe bus does not bottleneck the performance.

A second significant challenge is structuring algorithms such that they are suited to the highly threaded environment present on GPUs, GPU threads have tighter restrictions than CPU threads on how they may efficiently access memory and perform computation. Code which does not efficiently use the highly threaded GPU architecture is unlikely to provide a performance benefit over a typical CPU implementation.

Hardware Specific Functions

To produce the most efficient code developers may utilise "intrinsic" functions that are specific to a hardware architecture. These functions map to low level hardware instructions, for example, the vector instruction vfma. Explicit use of vector operations, such as the vfma operation, are a typical use case for intrinsic functions as a method to guarantee the output of a compiler. If a compiler is unwilling to automatically produce vector instructions for a section of code then an intrinsic function forces the compiler to produce specific instructions. The use of an intrinsic function imposes a restriction on the target architecture that an implementation can be executed upon, for example, the use of an intrinsic function for the Advanced Vector Extensions (AVX) instruction set produces an output that may only be executed on CPUs that feature AVX and hence is not portable to other CPUs or GPUs.

Similarly, codes written for Nvidia GPUs using CUDA are not portable to GPUs from other manufactures, such as AMD, or at the time of writing CPUs. On NVIDIA GPU hardware using intrinsic functions for inter-thread communication improves performance beyond what the compiler can produce, hence intrinsic functions are commonly used in CUDA code. As previously discussed, the intrinsic functions prevent an implementation from being compiled and executed on a different architecture in the absence of some translation process that removes or re-implements the intrinsic.

Levels Of Parallelism

A common theme of modern HPC architectures is the grouping of large numbers of identical components to increase the available computational power. At the level of CPU cores and GPU SMs there is SIMD data parallelism that applies the same operation to multiple

sets of data in parallel. Furthermore, within the compute cores themselves instruction level parallelism is exploited by re-ordering the flow of instructions to maximise the throughput of the compute core. Cores are grouped into CPUs which in turn form compute nodes that may also contain multiple accelerators, e.g. GPUs. Finally, identical compute nodes are connected via the high performance interconnects to form the HPC facility.

Each level of parallelism in the hierarchy of a HPC facility needs to be addressed efficiently by both algorithms and implementations to make efficient use of the computational resource. The number of levels of parallelism required to be efficient is a significant concern for the developers of HPC code especially in heterogeneous computing where multiple hardware architectures are used simultaneously.

Implications For Portable Performance

For a given computational task the most efficient algorithm and implementation for one hardware architecture is often not efficient on a second hardware architecture. Furthermore, at the point of design of a HPC code a developer can only be aware of current, past and near future hardware architectures. Typically, these constraints lead to code being developed which is either efficient on the most ubiquitous hardware available or is efficient on the leading performance hardware available. Our goal is to produce code for MD simulations which makes optimal use of modern hardware and is portable between hardware types.

Due to the technical expertise required to produce efficient code and the domain specific expertise required to design suitable algorithms MD code development requires knowledge from at least two different fields; the scientific domain to understand the problem and the computational domain to understand the hardware and associated programming methodologies. Regardless of the target architecture and the method employed to produce efficient performance critical code the production of a good quality implementation requires a significant investment of developer time for an initial release and for maintenance over the life of the software.

It is highly desirable to produce software which is portable between a wide range of hardware architectures and efficiently uses each computational resource. If an implementation is portable between different hardware architectures and efficiently utilises each hardware resource then it is referred to as "performance-portable". This performance-portable approach reduces redevelopment of existing functionality to ensure efficient use each hardware architecture.

Parallel Programming Models

We consider two parallel programming models; message passing and shared memory. Message passing is a form of inter-process communication. We consider an operating system process to consist of one or more threads that execute instructions and a region of memory which can only be accessed by this process. The memory region of a process can only be

accessed directly by the owning process and hence the transfer of data between processes requires one process to send data and another process to receive data. By using a Message Passing Interface (MPI) library the specific details of the inter-process communication are abstracted away from the programmer. Furthermore, MPI allows the send process to be on a completely separate compute node to the receiving process, the data is transferred over the high performance network. As these memory regions are distinct and can be on separate compute nodes this model is also referred to as a "distributed memory model".

A operating system process can contain more than one thread known as multithreading, each thread executes its own set of CPU instructions but they share the same memory region of the parent process, hence using multiple threads is a form of "shared memory" programming. Separate threads from a single process can execute instructions concurrently on separate CPU cores and can communicate through the shared memory. A very popular and portable method of multithreaded programming is the OpenMP application programming interface. The CUDA programming language used to programme Nvidia GPUs operates in a shared memory model. Each thread executed on the GPU cores can access data in the GPU memory, hence as with most multithreading models the programmer must take precautions to avoid race conditions. To utilise multiple GPUs, a robust and scalable approach is to launch one MPI process per GPU, this process acts as the controlling host for the GPU and allows communication between GPUs in separate nodes.

Using shared memory parallelism alongside distributed memory parallelism is beneficial when the overall efficiency of a MPI parallelised program is limited by inter-process communication. By reducing the number of MPI processes through increasing the use of shared memory parallelism the programmer aims to reduce the time spent communicating data. This combination of programming models is known as "hybrid" we employ hybrid parallelism with MPI as a distributed memory programming model combined with OpenMP as a shared memory programming model and refer to this combination as "MPI+OpenMP". For example, consider a compute node containing two separate E5-2650v2 CPUs (16 cores total), we could launch 1 MPI process per CPU and 8 OpenMP threads per MPI process which would result in 1 OpenMP thread per core.

1.3 Discussion Of Existing Libraries

1.3.1 Overview Of Existing Libraries

As computational chemistry and physics are active and established area of scientific computing there are many existing libraries created as tools for MD simulations. A subset of these codes are targeted towards particular areas of interest, for example, molecular biology. Generic MD libraries exist and provide access to collections of methods which potential users can use for simulation and analysis.

In the remainder of this section we will review some existing codes for atomistic simulations. Of particular interest for this project is the way the user controls the details

of the simulation. While for many codes it is relatively easy to change parameter values of a given inter-particle potential, more fundamental changes, such as modifying the functional form of the potential itself, may require changes to the source code. Successful source code modification requires knowledge of libraries that have significant code bases. Furthermore, the static nature of these libraries requires significant development work to target emerging architectures such as GPUs and Xeon Phi.

The interface for a large proportion of the libraries is through library specific configuration files, users determine values for the parameters available in the library and write these to a file. For example, a user may select a Lennard-Jones potential and specify their required values of ϵ and σ. Other common parameters are the temperature and the number of simulation steps. The library is then launched with a given configuration file and hopefully produces the results requested often with some history and logging information for the simulation.

The majority of the popular libraries offer parallel computation on distributed memory nodes using MPI which in some cases is combined with threading intra-node. For example, DL_POLY [41] provides a multi-process MPI executable designed to be launched with one process per core. The GROMACS [60] library supports hybrid MPI+OpenMP where a process may be launched per node which spawns enough threads to use all available cores on that node. GPU support is restricted to a subset of libraries with varying degrees of computation offloading e.g. LAMMPS and HOOMD-blue [59] [9]. There is less support for the Intel Xeon Phi architecture than for GPUs which is likely due to the infancy of the platform.

1.3.2 Library Comparisons

DL_POLY (full name DL_POLY_4) is a general purpose MD code currently targeted at conventional CPUs. Programmed in FORTRAN90, DL_POLY implements a spatial domain decomposition approach with communication between sub-domains handled with MPI. Users specify their desired functionality from a predefined set of options within a configuration file which is read in by the program and executed. DL_POLY is the only code we discuss that follows the classical approach of implementing a CPU only code with parallelism built using only MPI. Several other codes such as LAMMPS [63], GROMACS, NAMD [62] and AMBER [61] implement hybrid MPI+OpenMP parallelism to reduce both the communication cost of an iteration and the additional overhead at each iteration.

Within recent years the performance provided by GPUs has persuaded several organisations to port computationally intensive portions of code to the CUDA programming language for GPU acceleration. LAMMPS, GROMACS and AMBER claim varying levels of computation offloading but fall short of a complete offload to the GPU. For example, LAMMPS will offload the charge assignment and force interpolation portion of the Particle-Particle-Particle Mesh (P^3M) [35] method to a GPU [64] but uses the host for communication between nodes. Similarly to LAMMPS, HOOMD-blue will offload non-

bonded force calculations to multiple GPUs but will not offload the long range electrostatic interactions. Typically, users enable GPU offloading by enabling options in configuration files. In libraries where GPU support is provided through plugin systems users are expected to enable and compile the plugin into the library in a one time operation. Across all libraries there is minimal support for the Intel Xeon Phi, but there is evidence of development as NAMD and AMBER claim limited or experimental support.

User interaction varies on a per library basis, for example, LAMMPS provides an internal scripting environment where an input script specifies the operations performed by the library. The LAMMPS script in Appendix A.6 creates a uniform lattice of particles at a set density within a bounding domain. The script then assigns particle properties such as mass and velocity and sets a Lennard-Jones potential for all inter-particle interactions. Finally, the last line of the script instructs LAMMPS to integrate the system forward in time. In general, simulations share a similar flow of control to the LAMMPS example, an initial condition is formed which evolves over time using a specified integrator, output is either at the end of the simulation or periodically throughout the simulation.

Libraries such as OpenMM and HOOMD-blue provide interfaces to the library for external scripting languages like Python. An example script for HOOMD-blue is given in Appendix A.7 which expresses a similar simulation to the LAMMPS example but within a high level language. Custom non-bonded potentials are a feature which are implemented on a per library basis by either direct source code modification, loading tabulated values, writing custom plugins or some internal generation. DL_POLY, GROMACS, HOOMD-blue and NAMD will take as an input tabulated values of the potential energy and force magnitude for a given range of radii. The tabulated inputs are then interpolated by the libraries to be used internally in simulations. LAMMPS implements an approach where users write their extension as source code which is compiled into the library prior to runtime in an add-on style arrangement.

OpenMM provides high level functionality where a non-bonded interaction can be described as an analytic expression for the potential [21]. OpenMM then analytically computes the derivative of the expression to produce an analytic expression for the force as a function of radius. As the OpenCL execution model features runtime compilation for execution OpenMM compiles the analytic expression into an OpenCL kernel after an internal optimisation procedure [20]. OpenCL targets both CPU and GPU architectures hence this method of generation custom potential may be used for both the CPU and GPU. It should be noted that OpenMM does not support MPI and hence does not natively scale to multiple nodes, libraries such as GROMACS have successfully used OpenMM as a means to offload computation to GPUs for acceleration. Secondly, OpenMM provides only one abstraction layer, if users cannot describe their desired operation within this layer they must revert to source code modification.

1.3.3 Discussion And Conclusions

The primary aim of this project is not to achieve higher performance than existing production libraries, many of which are highly optimised, but to provide a flexible tool applicable to general tasks in the field of molecular simulation and analysis. No existing codes provide a framework to support code generation or user development to the extent provided by our abstraction. We have identified that by using one- and two-particle operations many MD algorithms can be described for simulations and analysis. The functionality of existing libraries can be replicated and extended in addition to increased efficiency and portability by using code generation.

OpenMM performs code generation specifically for custom forces between atomic objects and for custom integrators but does not target distributed memory architectures and does not provide a layered abstraction. Multiple existing libraries do provide high level interfaces to functionality but provide little or no access for user friendly modification or extension.

Often the simulation itself is not the only computation that a domain specialist wishes to perform. The simulation process provides data such as trajectories from the system of interest. This raw data has to be analysed to identify scientifically interesting behaviour. For example, analysis may investigate the local environment of each particle with local cluster analysis or investigate long range structure with the radial distribution function. Existing libraries provide limited on-the-fly analysis but do provide some predetermined methods of post processing. If a domain specialist requires something non-standard this is unlikely to be provided. Furthermore, with on-the-fly analysis the dynamics of the model can be modified during the simulation, implementing feedback from analysis measurements into simulation parameters would be extremely computationally expensive to implement with a post processing approach.

Complicated analysis may also require computation time comparable with the simulation itself, which poses problems for domain specialists not familiar with parallelisation and hardware. There does not appear to be any major library providing accelerator support for analysis computation. More generally, there is little support for parallel post processing analysis in either a shared memory or distributed memory setting.

CHAPTER 2

A SEPARATION OF CONCERNS BASED ABSTRACTION

2.1 PyOP2 And Firedrake: An Existing Approach

In this section we discuss the abstractions presented by PyOP2 [68] and Firedrake [67]. Together these two projects are an example of the concept that we apply to the field of MD simulation. PyOP2 provides a library to execute computational kernels on unstructured meshes. Firedrake uses PyOP2 as an internal component, alongside other frameworks such as PETSc [10], to provide a framework for computing numerical solutions to PDEs via finite element methods. We provide some background into Firedrake and PyOP2 as this combination of a high level interface built on a multiple level execution environment has proven to be successful in the finite element community. By applying the idea of using multiple levels of abstraction we produce a new abstraction tailored to MD, we aim to make the advances in software development now available in finite element software available in MD related fields.

The key problem that Firedrake solves with PyOP2 is that operations on elements in meshes in Finite Element Method (FEM) codes result in code which is complex but must be optimised for particular hardware to be efficient. As there are a limited number of people who are experts in FEM and HPC development Firedrake implements a separation of concerns approach which splits the overall problem into a number of smaller sub-problems which may then be tackled by field experts. The operations on elements are translated into computational kernels which can be automatically optimised by Firedrake for a specific hardware at runtime with acceptably low overhead.

The difference between FEM and MD is that in MD we loop over pairs of particles whereas in FEM Firedrake loops over elements in a mesh. Furthermore, unlike FEM local domains in MD will have differing numbers of particles, presenting memory and load balancing issues as particles move throughout the domain.

All mesh based codes to compute numerical solutions of PDEs require the execution of small computational kernels on each entity (cell, facet or vertex) of a mesh. For example,

a finite volume discretisation might increment the pressure value stored in each cell by summing up the fluxes on all facets which are touching this cell. Since topological relations between mesh entities are important, PyOP2 represents those in dedicated data structures and provides iterators over those.

PyOP2 starts with the idea that a mesh such as those found in scientific computations, e.g. computational fluid dynamics, can be represented by sets of entities connected together with maps. For example, a graph can be described by a set of vertices, a set of edges and a map which encodes which edges are connected to each vertex. The framework provides a Domain Specific Language (DSL) to represent sets of entities and maps between these sets. By decomposing a high level description into primitive objects optimisation can be performed in terms of these primitive objects without knowledge of the original high level problem. In the following example a triangle is represented by three vertices connected by three edges, the final line describes a map from edges to vertices. If we have edges e_0, e_1, e_2 and vertices v_0, v_1, v_2 we map $e_0 \to (v_0, v_1)$, $e_1 \to (v_1, v_2)$ and $e_2 \to (v_2, v_0)$.

```
vertices = op2.Set(3)
edges = op2.Set(3)
edges2vertices = op2.Map(edges, vertices, 2,
[[0, 1], [1, 2], [2, 0]])
```

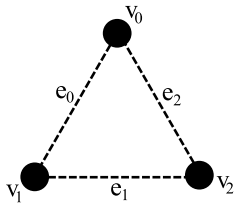

Figure 2-1: *Triangle described by three edges and three vertices, adapted from PyOP2 documentation [69].*

Considering the triangle above we assign a 2D position to each of the vertices. In the context of PyOP2 we attach data to elements of a set, i.e. we assign a tuple of two real numbers to each element in the set `vertices` by using the `op2.Dat` data structure. First we declare that each element of the set `vertices` has a two dimensional object associated with it by using a `op2.DataSet` object, secondly, a `op2.Dat` is created and connected with the created `op2.DataSet` object.

```
dvertices = op2.DataSet(vertices, dim=2)
coordinates = op2.Dat(dvertices,
              [[0.5, 0.9], [0.0, 0.0], [1.0, 0.0]],
              dtype=float)
```

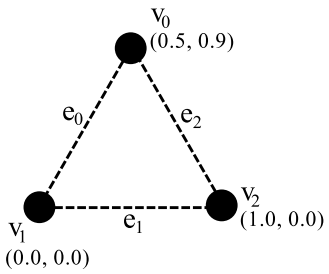

Figure 2-2: *Triangle from Figure 2-1 with positions added, adapted from PyOP2 documentation [69].*

Finally, suppose we wish to take each element in the set of vertices and apply a fixed translation along the vector $(1, 1)$. The translation operation is written as a C function and wrapped within a `op2.kernel` object with the intent that this kernel will be applied to

each vertex independently. As the application of the kernel to each vertex is independent the execution may occur in parallel, PyOP2 loops over each vertex in the set of vertices in parallel and at each vertex applies the translation kernel after a call to `op2.par_loop`.

```
translate = op2.Kernel(
        """void translate(double * coords) {
        coords[0] += 1.0;
        coords[1] += 1.0;
        }""", "translate")
op2.par_loop(translate, vertices,
coordinates(op2.RW))
```

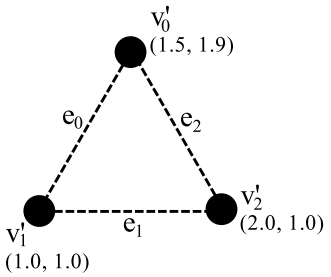

Figure 2-3: *Triangle from Figure 2-1 after translation along* $(1, 1)$*, adapted from PyOP2 documentation [69].*

PyOP2 generates C source code for a shared library that executes the kernel in the parallel loop. The C source is compiled into hardware instructions by the compiler after an optimisation stage that aims to maximise the efficiency of the resulting instructions. The compiler may only apply code transformations that do not impact the correctness of the output. Furthermore, the compiler only has the C source as an input and cannot apply any high level reasoning that exploits the structure of the algorithm. After applying code transformations, the compiler will emit hardware instructions that it heuristically determines are most efficient, vector instructions are often the most desirable for computation bound algorithms.

As the compiler has limited optimisation opportunities, the Firedrake implementation, in conjunction with PyOP2, produces highly efficient C source code by applying high level reasoning at the code generation stage. The Firedrake project includes the Two-Stage Form Compiler [36] that manipulates mathematical expressions and transforms generated code, this method produces a highly efficient C source for the host compiler.

The parallel loop invocation in Figure 2-3 indicates to the PyOP2 framework that the coordinate data is both read and written to by the kernel by using an access descriptor (`op2.RW`). Access descriptors are essential in determining which data is required to be communicated between private memory processes before kernel execution, the converse also applies, by only communicating required data unnecessary communication is avoided.

For example, if two `par_loops` are launched with kernels that both access the same variable in a read-only manner it can be reasoned that for the first loop execution any outstanding data dependencies must be resolved between MPI processes. For the second loop execution it can be reasoned that there are no outstanding data dependencies for this variable and hence communication between MPI processes is avoided.

2.2 An Abstraction For Particle Operations

Motivated by PyOP2, we present an abstraction for describing particle data and operations with particles, such as the methods we described in Section 1.1.3 and 1.1.4. This section is an adaptation of our published journal article [73]. We assume that we want to simulate and analyse a collection of $N \gg 1$ particles. Let each particle with global index $i \in \{0, 1, 2, \ldots, N-1\} \equiv \mathcal{N}$ have a set of properties π such that $\pi_r^{(i)}$ is the value of the r-th property on particle i. Each particle has exactly M properties, i.e. $r \in [0, M-1] \equiv \mathcal{M}$.

This abstract description of particle properties allows general per particle properties to be described. Described properties could correspond to physically relevant quantities such as position, momenta and charge. Furthermore, particle properties can correspond to higher-level information, for example, a description of the type of atom represented by the particle or to record which molecule an atom is currently a member of.

In addition to per particle properties there can be M^g *global* properties $\pi_{r^g}^g$ with $r^g \in [0, M^g - 1] \equiv \mathcal{M}^g$. Global properties allow the description and storage of quantities which are collective over the set of particles. For example, the total kinetic energy and potential energy of the system are global quantities. As with per particle properties, global properties are not restricted to physically motivated quantities and could record higher-level information, for example, the number of particles classified as members of FCC or HCP crystalline lattices.

We now describe how operations involving per particle properties and global properties are performed. We describe two looping operations; a *Particle Loop* which operates on individual particles and a *Particle Pair Loop* which operates on pairs of particles.

Definition 2.1. A *Particle Loop* is an operation which for each particle $i \in \mathcal{N}$ reads properties $\pi_r^{(i)}$ with $r \in \mathcal{M}_R \subset \mathcal{M}$ and writes properties $\pi_w^{(i)}$ with $w \in \mathcal{M}_W \subset \mathcal{M}$. The operation can also read global properties $\pi_{r^g}^g$ with $r^g \in \mathcal{M}_R^g \subset \mathcal{M}^g$ and write $\pi_{w^g}^g$ with $w^g \in \mathcal{M}_W^g \subset \mathcal{M}^g$ such that the final value of these global properties is independent of the looping order over the particles. This operation has an $\mathcal{O}(N)$ computational complexity as all particles are looped over once.

Example 2.2 below reads particle data and increments a global property. Mathematically the increment operation is associative hence the output quantity is independent of the order in which particles are looped over and reductions occur.

Example 2.2. *Kinetic energy calculation.* To calculate the total kinetic energy we loop over all particles i and add $\frac{1}{2}m^{(i)}\sum_{k=0}^{d-1}(v_k^{(i)})^2$ to the global variable K. The particle properties considered in this example are the mass $m^{(i)}$ and the three components $v_k^{(i)}$, $k = 0, 1, 2$ of the particle's velocity vector $\vec{v}^{(i)}$.

Particle Loops are not required to involve global properties and may only involve particle properties. *Particle Loops* that only involve particle properties are trivially independent of the order of execution.

Example 2.3. *Velocity update.* Given current particle velocities $\vec{v}_i$ and accelerations $\vec{a}_i$ the first operation in the Velocity Verlet algorithm (Alg. 1) performs $\vec{v}_i \leftarrow \vec{v}_i + \vec{a}_i \delta t/2$.

Inter-particle operations, such as computing force calculations, require computational loops over pairs of particles and access to particle properties from two distinct particles, which cannot be performed by a *Particle Loop*. Hence we define the *Particle Pair Loop* as a general approach to looping over pairs of particles in a manner that is appropriate for MD.

Definition 2.4. A *Particle Pair Loop* is an operation which for all particle pairs $(i, j) \in \mathcal{N} \times \mathcal{N}$ reads properties $\pi_r^{(i)}$ and $\pi_r^{(j)}$ with $r \in \mathcal{M}_R \subset \mathcal{M}$ and modifies properties $\pi_w^{(i)}$ with $w \in \mathcal{M}_W \subset \mathcal{M}$ such that the result is independent of the order of execution. The kernel can also read global properties $\pi_{r^g}^g$ with $r^g \in \mathcal{M}_R^g \subset \mathcal{M}^g$ and write $\pi_{w^g}^g$ with $w^g \in \mathcal{M}_W^g \subset \mathcal{M}^g$ such that the result does not depend on the order in which the loop is executed over all particle pairs. This operation has an $\mathcal{O}(N^2)$ computational complexity as all pairs of N particles are considered.

Example 2.5. *Force Calculation.* The most obvious example of a Particle Pair Loop is the force calculation. Here each particle has six relevant properties, namely the three entries of its position vector and the three entries of the force exerted on the particle by all other particles. For each particle pair the total force on the first particle is incremented by the interaction force $\vec{f}(\vec{r}^{(i)}, \vec{r}^{(j)})$ which depends on the relative position of the particles, i.e. the three position properties $r_k^{(i)}$ for $k = 0, 1, 2$ are read and the three force properties $F_k^{(i)}$ are incremented as $F_k^{(i)} \leftarrow F_k^{(i)} + f_k(\vec{r}^{(i)}, \vec{r}^{(j)})$.

The *Particle Pair Loop* considers all possible pair of particles and has a very general definition. As discussed in Sections 1.1.3 and 1.1.4, it is highly common for particles which are spatially well separated to not interact, hence it is computationally advantageous to formally define a variant of *Particle Pair Loop* that only requires pairs of particles which are spatially near to each other.

Definition 2.6. A *Local Particle Pair Loop* is a Particle Pair Loop that is guaranteed to include all pairs of particles which are separated by a distance that is less than or equal to a cutoff distance r_c. An implementation of a *Local Particle Pair Loop* may also loop over additional pairs of particles that are separated by a distance that is greater than r_c. The algorithms described in Chapter 3 demonstrate how this operation can be performed with an $\mathcal{O}(N)$ computational complexity.

Example 2.7. *Truncated Force Calculation.* Consider an inter-particle potential with a functional form that allows truncation at some cutoff radius r_c, for example, the Lennard-Jones potential in section 1.1.3. A *Local Particle Pair Loop* considers only pairs of particles for which it is known in advance that the interaction could be non-zero.

Example 2.8. *Local environment.* Suppose that each atom can be in one of two possible states. For every atom we want to count the number of other atoms in the same state

which are up to a distance r_c away. In this case each particle would have five properties, namely the three entries of the position vector, the state of the atom and the number of atoms in the same state in the local environment. For each pair of atoms the Local Particle Pair kernel would first check whether they are less than r_c apart by calculating the distance $|\vec{r}^{(i)} - \vec{r}^{(j)}|$ between the particle positions. If this is the case, and both particles are in the same state, the counter for the number of same-state atoms is increased. The inter-particle distances are compared with r_c as a *Local Particle Pair Loop* is allowed to execute the kernel for particle pairs that are separated by more than r_c.

Newton's Third Law

For the vast majority of physically realistic inter-particle potentials the interaction between a pair of particles produces inter-particle forces which are equal in magnitude but opposite in direction, this is known as Newton's Third Law. It should be noted that there exist areas of research interested in particle simulations where Newton's Third Law is not applied [42].

In principle this effect can be exploited to reduce the computational work of a *Particle Pair Loop* by considering each pair of particles once, the *Particle Pair Loop* could compute the magnitude of the exerted force once and update the forces of both participating particles. The iteration set over the $N(N-1)/2$ ordered pairs with $i < j$ contains half the elements of the iteration set of $N(N-1)$ unordered pairs and hence naively one could expect a speedup of a factor two.

A modification to Definition 2.4 to allow Newton's Third Law for a pair of particles (i, j) would enable the *Particle Pair Loop* to both read and write to the properties $\pi_s^{(j)}$. However as we discuss in Section 3.1 this modification is not always advantageous and in particular this modification causes significant implementation challenges on certain modern HPC hardware.

2.3 Abstraction Implementation

2.3.1 Domain Specific Language

We present a Python-embedded Domain Specific Language (DSL) designed to facilitate the implementation of algorithms written in our abstraction. As in PyOP2 the purpose of the DSL is to provide a high-level programming environment within which algorithms are written using the data structures and looping mechanisms of the abstraction. The previous section identifies and defines the data structures and looping methods which are crucial to both MD simulation and MD related analysis. We describe an interface to data structures and looping mechanisms which are sufficient to implement the abstraction. The abstraction requires data structures to store per particle properties $\pi_r^{(i)}$ and global properties $\pi_{r^g}^g$ alongside looping mechanisms for the *Particle Loop* and *(Local) Particle Pair Loops*. In our framework this interface is provided by a code generation system which generates efficient machine code and is described in Chapter 3. We focus on the

implementation of the *Local Particle Pair Loop* and not the more general *Particle Pair Loop* as in practice the first is vastly more relevant for simulations.

Data Structures

We now describe the data structures we implement to write algorithms in terms of our abstraction. These data structures separate the user from low level considerations such as memory management and MPI communication whilst providing the user with a familiar interface to the underlying data, a summary of the data structures we define is given in Table 2.1.

Particle properties $\pi_r^{(i)}$ are represented by instances of a `ParticleDat` class. This class is a wrapper around a 2D numpy [78] array where the properties of particle i populate row i in the array such that the array index (i, r) stores $\pi_r^{(i)}$. For global properties which are not associated with any particular particle we provide the `ScalarArray` and `GlobalArray` classes which are wrappers around 1D numpy arrays, unlike `ScalarArray` a `GlobalArray` object performs global reductions automatically.

We allow properties to be split across multiple `ParticleDat` instances, which can be named by the user. Splitting properties into multiple `ParticleDat` instances allows for properties of different data types and allows for further optimisation in the underlying framework. Similarly, global properties can be split across multiple `ScalarArray` and `GlobalArray` instances. For example, consider a simulation where each particle i has a position $\vec{r}_i \in \mathbb{R}^3$, velocity $\vec{v}_i \in \mathbb{R}^3$, acceleration $\vec{a}_i \in \mathbb{R}^3$ and a species indicator $s_i \in \mathbb{N}$. Furthermore, suppose that we wish to compute and store the kinetic energy $\mathcal{K} \in \mathbb{R}$ and potential energy $\mathcal{U} \in \mathbb{R}$. This configuration of local and global properties would be implemented as shown in Listing 2.1

Listing 2.1: *Data structure initialisation*

```
r = ParticleDat(ncomp=3, dtype=c_double)
v = ParticleDat(ncomp=3, dtype=c_double)
a = ParticleDat(ncomp=3, dtype=c_double)
s = ParticleDat(ncomp=1, dtype=c_int)
KE = GlobalArray(ncomp=1, dtype=c_double)
PE = GlobalArray(ncomp=1, dtype=c_double)
```

The wrapped numpy arrays can be accessed through the Python "getitem" and "setitem" methods which automatically mark particle data as "dirty" if the internal data has been directly modified by the user. This marking process is an important process to maintain consistency between memory regions in distributed memory programming models. We later present a parallel implementation based on a domain decomposition approach where the simulation domain is divided into disjoint regions which are assigned to MPI processes. Each MPI process "owns" the assigned region of the simulation domain and the particles

contained within this region and stores a local copy of particle data from neighbouring regions. By marking the `ParticleDat` as "dirty" all local copies of these particle properties are marked as invalid to ensure that up-to-date values are communicated between neighbouring processes if these particular properties are to be accessed in a *(Local) Particle Pair Loop*. This communication is automatically performed by the `ParticleDat` when required.

The MPI process which owns a particular particle is determined by which sub-domain of the simulation domain the particle resides in. Furthermore, the time taken to communicate particle data between neighbouring processes has a large impact on the parallel efficiency of the implementation, to ensure that this communication is as efficient as possible particle data is only communicated if the particle is sufficiently near to the sub-domain boundary. Hence the implementation requires knowledge of which `ParticleDat` contains the particle positions for book-keeping and efficiency reasons, particle positions are stored in the `PositionDat` class which is a sub-class of `ParticleDat` and is identical in all but name.

The `ParticleDat` "getitem" and "setitem" methods provide a transparent interface to particle data when the data is stored in GPU memory. A GPU `ParticleDat` instance will automatically copy data between host memory and device memory without user prompting and make this data available in a numpy array. By providing a consistent interface a Python script can implement an algorithm that accesses particle data through the `ParticleDat` objects that can be executed on a range of hardware with minimal modification. The hardware architecture is chosen in the Python script by setting aliases for the data structures as shown in Listing 2.2.

Listing 2.2: *Switching between CPU and GPU implementation*

```
import ppmd as md
# Set USE_CUDA to True or False
if not USE_CUDA:
    # define Data.* to refer to host (not CUDA)  data structures
    Data = md.data
    State = md.state.State

    # set aliases that refer to host looping methods
    ParticleLoop = md.loop.ParticleLoop
    PairLoop = md.pairloop.PairLoopNeighbourListNS
else:
    Data = md.cuda.cuda_data
    State = md.cuda.cuda_state.State

    # set aliases that refer to CUDA looping methods
    ParticleLoop = md.cuda.cuda_loop.ParticleLoop
    PairLoop = md.cuda.cuda_pairloop.PairLoopNeighbourListNS

# Define convenient aliases for the data structures
PositionDat = Data.PositionDat
ParticleDat = Data.ParticleDat
ScalarArray = Data.ScalarArray
GlobalArray = Data.GlobalArray
```

We provide the `State` class to group together a set of `ParticleDat` instances along with a domain and a boundary condition. In a MD simulation it is expected that particles will move between the sub-domains formed by the domain decomposition approach. When a particle moves from a sub-domain into a neighbouring sub-domain the ownership of that particle is transferred along with all data associated with that particle. As each sub-domain is owned by a separate MPI process communication must occur to enact this transfer. In our implementation the `State` class automatically generates code to efficiently pack, transfer and unpack the data associated with all particles that transfer between sub-domains.

Listing 2.3 demonstrates how domain, boundary conditions and particle data are added to a `State` instance. We begin by creating a `State` instance `A` which is assigned a domain and a domain boundary condition. Given a domain and a `PositionDat` the framework can apply a domain decomposition approach to assign sub-domains to MPI ranks, furthermore, the positions of particles determines the parent sub-domain and hence the owning MPI rank. The boundary condition determines exactly what behaviour occurs at the edge of the domain, a periodic boundary condition instructs the `State` object that particles that leave the simulation domain should be suitably "wrapped" around the domain. After the state `A` is assigned a boundary condition `ParticleDat` instances are added to the state.

If a particle property is added to `A` as a `ParticleDat` then each particle in `A` is given the property. Finally, in Listing 2.3 we add `GlobalArray` instances to store system energies in, these do not need to be associated with a state object.

Listing 2.3: *State initialisation example*

```
# Create a State instance to combine further data structures
A = State()

# Set the number of particles
A.npart = 10000

# Set the domain and boundary condition
A.domain = domain.BaseDomainHalo(extent=(10., 10., 10.))
A.domain.boundary_condition = domain.BoundaryTypePeriodic()

# Add a PositionDat instance for particle positions
A.r = PositionDat(ncomp=3, dtype=c_double)

# Add further ParticleDat instances
A.v = ParticleDat(ncomp=3, dtype=c_double)
A.a = ParticleDat(ncomp=3, dtype=c_double)
A.s = ParticleDat(ncomp=1, dtype=c_int)

# GlobalArray instances to store energy
KE = GlobalArray(ncomp=1, dtype=c_double)
PE = GlobalArray(ncomp=1, dtype=c_double)
```

Kernels And Constants

Both *Particle Loops* and *(Local) Particle Pair Loops* execute a computational kernel over either particles or pairs of particles. This computational kernel is implemented in a section of C code that implements the required operation. Particle properties are accessed in the C code through the syntax `<symbol>.<pair_index>[<component>]` where `<symbol>` is a user defined string that identifies the `ParticleDat` that holds the property, `<pair_index>` is either `i` or `j` and `<component>` defines which component of the `ParticleDat` should be accessed. In a *Particle Loop* operation `<pair_index>` can only be `i` as these kernels are applied once to all particles in isolation, i.e. there is no second particle to be indexed by `j`. Access to global properties stored in *ScalarArray* or *GlobalArray* objects is provided through the C identifier `<symbol>[<component>]` where `<symbol>` is a user specified identifier and `<component>` indexes into the array. For example:

- `r.i[0] += 1.0;`, increment by one the first component (0 indexing) of a `ParticleDat` labeled in the kernel as `r` in either a *Particle Loop* or *(Local) Particle Pair Loop*.

- `double vtmp = v.j[2];`, read the third component of a `ParticleDat` labeled in the kernel as `v` in a *(Local) Particle Pair Loop*.
- `KE[0] += 4.0;`, increment a *ScalarArray* or *GlobalArray* labeled as `KE` by four.

An example of a *Particle Loop* kernel is given by the final step of Algorithm 1 which updates particle velocities `v` by using particle accelerations `a`, this operation is implemented in the kernel in Listing 2.4. Furthermore, we update the kinetic energy `KE` using the new velocities, we assume the particles have unit mass. This kernel contains a constant `hdt` which is substituted at the code generation stage (or directly in the Python script) for the numerical value of $\delta t/2$. Instances of the `Constant` class hold the numerical value of a symbol and are passed into the constructor of a `Kernel`.

Listing 2.4: *Final step of Alg. 1 implemented in a C kernel.*

```
vv_kernel = Kernel(
    name='VV_example',
    code = """
    v.i[0] += hdt * a.i[0];
    v.i[1] += hdt * a.i[1];
    v.i[2] += hdt * a.i[2];
    KE[0] += 0.5 * (v.i[0]*v.i[0] + v.i[1]*v.i[1] + v.i[2]*v.i[2]);
    """,
    constants=(Constant('hdt', 0.0001),)
)
```

The Lennard-Jones potential in Equation (1.15) is a classic example of an inter-particle potential that is typically truncated at a cutoff radius r_c. The potential and force can be efficiently computed by writing the potential and the first derivative as

$$U_{\mathrm{LJ}}(r) = 4\epsilon \left(\frac{\sigma}{r}\right)^6 \left[\left(\frac{\sigma}{r}\right)^6 - 1\right] + s_{r_c}, \tag{2.1}$$

$$\text{where} \quad s_{r_c} = 4\epsilon \left(\frac{\sigma}{r_c}\right)^6 \left[\left(\frac{\sigma}{r_c}\right)^6 - 1\right], \tag{2.2}$$

$$-\frac{1}{r}\frac{\partial U_{\mathrm{LJ}}(r)}{\partial r} = \frac{48\epsilon}{\sigma^2} \left(\frac{\sigma}{r}\right)^8 \left[\left(\frac{\sigma}{r}\right)^6 - \frac{1}{2}\right]. \tag{2.3}$$

By writing the force magnitude as in Equation (2.3) we incorporate a $1/r$ term to normalise the direction vector the force acts along and we have written the magnitude in even only powers of $1/r$. Avoiding odd powers is a optimisation to avoid a square root evaluation, which is expensive.

Listing 2.5: *Lennard-Jones potential implemented in a C kernel. With substituted constants* `CV` $= 4\epsilon$, `CF` $= -48\epsilon/\sigma^2$ *and* `cutoff_shift` $= s_{r_c}$.

```
lj_kernel = Kernel(
    name='lj_example',
    """
    const double R0 = r.j[0] - r.i[0];
    const double R1 = r.j[1] - r.i[1];
    const double R2 = r.j[2] - r.i[2];
    const double r2 = R0*R0 + R1*R1 + R2*R2;

    const double r_m2 = sigma2/r2;
    const double r_m4 = r_m2*r_m2;
    const double r_m6 = r_m4*r_m2;
    PE[0] += (r2 < rc2) ? 0.5*CV*(r_m6-1.0)*r_m6 + cutoff_shift : 0.0;
    const double r_m8 = r_m4*r_m4;
    const double f_tmp = CF*(r_m6 - 0.5)*r_m8;
    a.i[0] += (r2 < rc2) ? f_tmp*R0 : 0.0;
    a.i[1] += (r2 < rc2) ? f_tmp*R1 : 0.0;
    a.i[2] += (r2 < rc2) ? f_tmp*R2 : 0.0;
    """,
    constants=(Constant('CF', ...), Constant('CV', ...),
        Constant('cutoff_shift', ...), Constant('rc2',rc*rc))
)
```

Manually writing C kernels provides the user with flexibility in describing the operation performed at the cost of increased complexity and scope for mistake. In principle the kernel can be generated from a higher level language or from a mathematical expression that describes the potential or operation, we do not discuss in any further detail the generation of kernels from higher level languages.

Access Descriptors

In addition to the kernel itself the user must indicate which C symbol corresponds to which Python data structure and state exactly how the kernel accesses the data. For example, in the Velocity Verlet kernel in Listing 2.4 particle accelerations are accessed in a read-only manner, particle velocities are incremented and the kinetic energy is incremented. Hence the corresponding accesses descriptors are; `READ` for accelerations, `INC` (increment) for velocities and `INC` for kinetic energy. The remaining possible access descriptors are; `RW` (read and write), `INC_ZERO` (increment after setting values to zero beforehand) and `WRITE` (data is written to). A summary of permissible access descriptors is given in Table 2.2, data structures may only support a subset of all access descriptors.

The access descriptors indicate to the implementation how a kernel will access data and hence determine exactly what code should be generated to execute the kernel. For example, if values in a data structure are only read from then the underlying data can

Description	Python Class
Collection of properties for all particles with d components per particle. All values are initialised to x_0 when the object is created.	`ParticleDat(ncomp=`d`, dtype=c_double/..., initial_value=`x_0`)`
Specialisation of `ParticleDat` for particle positions (see Section 2.3.1).	`PositionDat(ncomp=`d`, dtype=c_double/..., initial_value=`x_0`)`
Global property (not specific to individual particles) with d components; values are initialised to y_0.	`ScalarArray(ncomp=`d`, dtype=c_double/..., initial_value=`y_0`)`
Global property with d components; values are initialised to 0. Only permissible access types are `READ`, `INC`, `INC_ZERO`. Unlike `ScalarArray` objects entries are consistent across all MPI ranks, i.e. MPI reductions are automatically performed.	`GlobalArray(ncomp=`d`, dtype=c_double/...,)`
Numerical constant which is replaced by its specific value in kernel, i.e. the string L is replaced by the numerical value x in the generated C-code.	`Constant(name=`L`, value=`x`)`
Kernel object which can be used in one of the looping classes defined in Table 2.3. The C-source code is given as a string S and any numerical constants C_1, C_2, ... can be passed in as a list of `Constant` objects.	`Kernel(name=`L`, code=`S`, constants=(`$C_1, C_2, \ldots,$`) )`

Table 2.1: *Fundamental data classes of the DSL*

be marked as constant to enable more efficient code to be generated. It is essential that the code generation system creates looping code that is correct for the written kernel, the access descriptors indicate exactly what code should be generated to ensure correct execution of the kernel.

Furthermore, if particle data is only read from in a kernel then all copies of the read-only data that were made before the loop execution remain valid after the completion of the loop. If a kernel writes to particle data then the data is automatically marked as dirty after loop execution, this invalidates all copies of this data to ensure consistency.

As *(Local) Particle Pair Loops* operate on pairs of particles it is expected that due to the domain decomposition approach the two particles will often reside on different MPI processes, this scenario is the main reason particle data is duplicated. The transfer of this particle data is relatively expensive in comparison to computation and is greatly reduced by using access descriptors to only transfer required data, for example a kernel may only access particle positions, in which case there is no reason to communicate other particle data such as charge. Secondly, if copies are not invalidated by either an access descriptor or use of the "setitem" method the copied data can be reused for multiple kernel launches.

Description	Access Descriptor
Read-only access	`access.READ`
Write-only access	`access.WRITE`
Read and write access	`access.RW`
Incremental access	`access.INC`
Incremental access, initialise to zero	`access.INC_ZERO`

Table 2.2: *Supported access descriptors*

Particle Loops

Given a `Kernel` object that implements a operation for a *Particle Loop*, as in Listing 2.4, a `ParticleLoop` object is responsible for executing the kernel over all particles as in Definiton 2.1. A `ParticleLoop` is constructed with a `Kernel` object and a Python dictonary that maps the C symbols used in the kernel to `ParticleDat` instances along with access descriptors. In Listing 2.6 a `ParticleLoop` is constructed to execute the Velocity Verlet kernel defined in Listing 2.4. The created `ParticleLoop` object defines an `execute` method that when called executes the loop, hence an operation can be executed multiple times without re-construction of the `ParticleLoop`.

Listing 2.6: *Particle Loop example with data structures defined in Listing 2.3 and Velocity Verlet kernel defined in Listing 2.4.*

```
# Create ParticleLoop instance
vv_particle_loop = ParticleLoop(
    kernel=vv_kernel,
    dat_dict={
        'v': A.v(INC),
        'a': A.a(READ),
        'KE': KE(INC_ZERO)
    }
)
# Execute the loop once
vv_particle_loop.execute()
```

Particle Pair Loops

In principle a user could wish to execute an operation over all pairs of particles in the simulation. An execution of a kernel over all pairs will exhibit a $\mathcal{O}(N^2)$ computational complexity, far higher than the execution of a kernel over all pairs of particles which are within a cutoff radius of each other which exhibits a computational complexity $\mathcal{O}(N)$. Furthermore, the high computational complexity of the all-to-all particle pair loop becomes inefficient and prohibitively expensive for a moderate number of particles. Hence we focus

on the *Local Particle Pair Loop* which applies a kernel to pairs of particles which are within a cutoff $r_c =$ `rc` of each other.

The `PairLoop` behaves identically to the `ParticleLoop` object with the addition that to apply a *Local Particle Pair Loop* a cutoff radius is required and should be passed to the constructor of the `PairLoop`. In Listing 2.7 a `PairLoop` is constructed to execute the Lennard-Jones kernel defined in Listing 2.5.

Listing 2.7: *Pair Loop example with data structures defined in Listing 2.3 and Lennard-Jones kernel defined in Listing 2.5.*

```
# Create ParticleLoop instance
lj_pairloop = PairLoop(
    kernel=lj_kernel,
    dat_dict={
        'r': A.r(READ),
        'a': A.a(INC_ZERO),
        'PE': PE(INC_ZERO)
    },
    shell_cutoff=rc
)
# Execute the loop once
lj_pairloop.execute()
```

Description	Python Class
Execute `Kernel` object k for all particles and modify particle data (`ParticleDat`, `PositionDat`, `ScalarArray` or `GlobalArray` objects) d_1, d_2, ... Each particle data object d_i can be accessed via the corresponding label L_i and has access descriptor A_i defined in Table 2.2.	`ParticleLoop(kernel=`k`, dat_dict={`L_1`:`$d_1(A_1)$`, `L_2`:`$d_2(A_2)$`, ...} )`
Same as `ParticleLoop`, but execute the kernel over all *pairs* of particles.	`PairLoop(kernel=`k`, dat_dict={`L_1`:`$d_1(A_1)$`, `L_2`:`$d_2(A_2)$`,...})`

Table 2.3: *Fundamental looping classes of the DSL*

2.3.2 Further Examples

These examples are excerpts from the published article titled "A Domain Specific Language for Performance Portable Molecular Dynamics Algorithms" [73]. They demonstrate that our abstraction is not only limited to force calculations, but can also be used to express more complicated analysis algorithms.

Bond Order Analysis

The BOA method introduced in Section 1.1.4 is motivated by the multipole expansion of the local environment of a particle. It computes the quantities $Q_\ell^{(i)}$, $\ell \in \{4, 5, 6\}$ for each particle i where

$$Q_\ell^{(i)} = \sqrt{\frac{4\pi}{2\ell+1} \sum_{m=-\ell}^{+\ell} |q_{\ell m}^{(i)}|^2}, \tag{2.4}$$

where

$$q_{\ell m}^{(i)} = \frac{1}{|\mathcal{N}(i)|} \sum_{j \in \mathcal{N}(i)} Y_\ell^m(\hat{\vec{r}}_{ij}), \tag{2.5}$$

$$\hat{\vec{r}}_{ij} = \frac{\vec{r}_i - \vec{r}_j}{|\vec{r}_i - \vec{r}_j|}, \tag{2.6}$$

$\mathcal{N}(i)$ is the set of neighbours of particle i and Y_l^m are the spherical harmonics as defined in Section 1.1.4. $q_{lm}^{(i)}$ encodes the multipole moments of the angular part of the density distribution formed by the neighbours of particle i. The order parameters $Q_\ell^{(i)}$ can be calculated with the two loops shown in Algorithms 2 and 3. The first local particle pair loop (Algorithm 2) calculates the number of neighbours $\nu_{\text{nb}}^{(i)} = |\mathcal{N}(i)|$ and the moments

$$\tilde{q}_{\ell m}^{(i)} = \sum_{j \in \mathcal{N}(i)} Y_\ell^m(\hat{\vec{r}}^{(i,j)}) \quad (= \nu_{\text{nb}}^{(i)} q_{\ell m}^{(i)}) \tag{2.7}$$

for $m = -\ell, \ldots, +\ell$ for each atom i; those quantities are stored in two `ParticleDat`s. The particle loop in Algorithm 3 uses $\nu_{\text{nb}}^{(i)}$ and $\tilde{q}_{\ell m}^{(i)}$ to calculate the $Q_\ell^{(i)}$ according to Equation (2.4); the result is stored in a third `ParticleDat`.

Algorithm 2: BOA Local Particle Pair Loop I.

Data: particle positions $\vec{r}^{(i)}$ [`READ`]
Result: moments $q_{\ell m}^{(i)}$ [`INC_ZERO`], neighbour counts $\nu_{\text{nb}}^{(i)}$ [`INC_ZERO`]
for *pairs* (i, j) **do**
  if $|\vec{r}^{(i)} - \vec{r}^{(j)}| < r_c$ **then**
    $\hat{\vec{r}}^{(i,j)} \leftarrow (\vec{r}^{(i)} - \vec{r}^{(j)})/|\vec{r}^{(i)} - \vec{r}^{(j)}|$
    for $m = -\ell, \ldots, +\ell$ **do**
      $\tilde{q}_{\ell m}^{(i)} \leftarrow \tilde{q}_{\ell m}^{(i)} + Y_\ell^m(\hat{\vec{r}}^{(i,j)})$
    end
    $\nu_{\text{nb}}^{(i)} \leftarrow \nu_{\text{nb}}^{(i)} + 1$
  end
end

Common Neighbour Analysis

The Common neighbour analysis (CNA) introduced in Section 1.1.4 is an example of a highly non-trivial algorithm that can be implemented within the abstraction with relative

Algorithm 3: BOA Particle Loop II.

Data: moments $\tilde{q}_{\ell m}^{(i)}$ [`READ`], number of local neighbours $\nu_{\mathrm{nb}}^{(i)}$ [`READ`]
Result: $Q_\ell^{(i)}$ [`WRITE`]
for $m = -\ell, \ldots, +\ell$ **do**
  $q_{\ell m}^{(i)} \leftarrow \tilde{q}_{\ell m}^{(i)} / \nu_{\mathrm{nb}}^{(i)}$
end
$Q_\ell^{(i)} \leftarrow \sqrt{\frac{4\pi}{2\ell+1} \sum_{m=-\ell}^{+\ell} |q_{\ell m}^{(i)}|^2}$

ease. To implement the CNA algorithm in our abstraction we proceed in two steps: For each atom i we first calculate all directly and indirectly bonded atoms. The set $\mathcal{E}_d^{(i)}$ describes the direct bonds which exist between atom i and all other atoms within a cutoff radius r_c. The indirect bonds in the local environment are collected in $\overline{\mathcal{E}}^{(i)}$ (see Figure 2-4), the indirect bonds of atom i are the direct bonds of atoms which are themselves directly bonded to i:

$$\begin{aligned} \mathcal{E}_d^{(i)} &= \big\{(i,v) : v \in \mathcal{N}, |\vec{r}^{(i)} - \vec{r}^{(v)}| < r_c\big\} \\ \overline{\mathcal{E}}^{(i)} &= \big\{(v,w) : v,w \in \mathcal{N}, |\vec{r}^{(v)} - \vec{r}^{(w)}| < r_c, \\ &\qquad |\vec{r}^{(i)} - \vec{r}^{(v)}| < r_c\big\} \end{aligned} \tag{2.8}$$

Since some of the indirect bonds are counted twice in $\overline{\mathcal{E}}^{(i)}$, the set $\mathcal{E}^{(i)}$ is an ordered representation of the same bonds:

$$\mathcal{E}^{(i)} = \big\{(v,w) : (v,w) \in \overline{\mathcal{E}}^{(i)}, v < w\big\} \subset \overline{\mathcal{E}}^{(i)}. \tag{2.9}$$

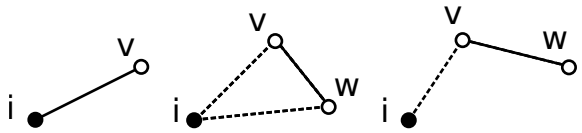

Figure 2-4: *Example of direct (left) and indirect (centre and right) bonds as described by the sets $\mathcal{E}_d^{(i)}$, $\overline{\mathcal{E}}^{(i)}$ and $\mathcal{E}^{(i)}$ in Equations (2.8) and (2.9). The bond (v,w) in the central diagram would be counted twice in $\overline{\mathcal{E}}^{(i)}$ but only once in $\mathcal{E}^{(i)}$.*

All atoms are assigned a global index in $\mathcal{N} = \{0, \ldots, N-1\}$ and the set of all neighbours of particle i is denoted $\mathcal{N}(i)$, i.e. all other atoms j which are within the cutoff r_c, hence if $j \in \mathcal{N}(i)$ then a direct bond exists between i and j. In a second step we loop over all pairs (i,j) of atoms and calculate the sets

$$\begin{aligned} \mathcal{C} &= \mathcal{N}(i) \cap \mathcal{N}(j) \\ \mathcal{E} &= \{(v,w) : v,w \in \mathcal{C}, v < w\} \subset \mathcal{E}^{(i)} \cap \mathcal{E}^{(j)}. \end{aligned} \tag{2.10}$$

$\mathcal{C}$ is the set of common neighbours and $\mathcal{E}$ is the set of common neighbour bonds. To avoid double counting we consider ordered bonds $(v, w) \in \mathcal{E}^{(i)}$ such that $v < w$. Together the two sets $\mathcal{C}$ and $\mathcal{E}$ define the graph $\mathcal{G}$ where common neighbours $\mathcal{C}$ are the graph vertices and bonds between the common neighbours $\mathcal{E}$ are the graph edges. The first two entries of the triplet $(n_{\text{nb}}, n_{\text{b}}, n_{\text{lcb}})$ can be calculated directly as $n_{\text{nb}} = |\mathcal{C}|$ and $n_{\text{b}} = |\mathcal{E}|$.

To calculate the size of all subgraphs $\mathcal{G}' \subset \mathcal{G}$, a random node $v \in \mathcal{G}$ is chosen. The size of the subgraph $\mathcal{G}'$ such that $v \in \mathcal{G}'$ is obtained with a breadth-first traversal of the connected component containing v, removing all visited nodes from $\mathcal{G}$ in the process. This is repeated until all nodes have been removed, thus calculating the size of all subgraphs $\mathcal{G}' \subset \mathcal{G}$. The computation of the maximal cluster size $n_{\text{lcb}} = \max_{\mathcal{G}' \subset \mathcal{G}}\{|\mathcal{G}'|\}$ with this method is shown explicitly in Algorithm 25 in A.1.

The CNA algorithm can be implemented with three Local Particle Pair Loops. We require the following particle properties and corresponding `ParticleDats`:

$\vec{r}$ (`ncomp=3, dtype=c_double`) $\vec{r}^{(i)}$ stores the position of particle i.

G (`ncomp=1, dtype=c_long`) $\vec{G}^{(i)}$ stores the unique global index particle i.

$v_{\mathbf{nb}}$ (`ncomp=1, dtype=c_long`) $\nu_{\text{nb}}^{(i)}$ stores the number of neighbours of particle i, i.e. $\nu_{\text{nb}}^{(i)} = |\mathcal{N}(i)|$; this is the number of red particles in the inner circle in Figure 2-5.

ν_{b} (`ncomp=1, dtype=c_long`) Number of bonds in the local environment. $\nu_{\text{b}}^{(i)} = |\mathcal{E}_d^{(i)} \cup \overline{\mathcal{E}}^{(i)}|$ counts the directly bonded neighbours of a particle plus the number of indirect bonds defined in Equation (2.8).

E (`ncomp=`$2\nu_{\text{b}}^{(\max)}$`, dtype=c_long`) Array representation of the set $\mathcal{E}_d^{(i)} \cup \overline{\mathcal{E}}^{(i)}$ defined in Equation (2.8). Two consecutive entries $E_{2k}^{(i)}, E_{2k+1}^{(i)}$ represent a bonded pair in the local environment of particle i, i.e. one of the links shown in Figure 2-5. The entries of $E^{(i)}$ are arranged as follows:

- $(E_{2k}^{(i)}, E_{2k+1}^{(i)}) = (G^{(i)}, G^{(j)})$ with $j \neq i$ for $0 \leq k < \nu_{\text{nb}}^{(i)}$
- $(E_{2k}^{(i)}, E_{2k+1}^{(i)}) = (G^{(j')}, G^{(j'')})$ with $j' \neq i$, $j'' \neq i$ for $\nu_{\text{nb}}^{(i)} \leq k < \nu_{\text{b}}^{(i)}$

In other words, the first $\nu_{\text{nb}}^{(i)}$ tuples represent the bonds in $\mathcal{E}_d^{(i)}$ and are shown as red (solid) lines in Figure 2-5. The remaining $\nu_{\text{b}} - \nu_{\text{nb}}$ tuples describe the set $\overline{\mathcal{E}}^{(i)}$ and correspond to the blue (dashed) lines. The static size $\nu_{\text{b}}^{(\max)}$ of the list has to be chosen sufficiently large, i.e. $\nu_{\text{b}}^{(\max)} \geq \max_i\{\nu_{\text{b}}^{(i)}\}$.

T(`ncomp=`$3\nu_{\text{nb}}^{(\max)}$`, dtype=c_long`) Stores the triplets $(n_{\text{nb}}, n_{\text{b}}, n_{\text{lcb}})$ such that the triplet $(T_{3j}^{(i)}, T_{3j+1}^{(i)}, T_{3j+2}^{(i)})$ is $(n_{\text{nb}}, n_{\text{b}}, n_{\text{lcb}})$ for the j-th bonded neighbour of particle i. The number of components $\nu_{\text{nb}}^{(\max)}$ has to be chosen such that $\nu_{\text{nb}}^{(\max)} \geq \max_i\{\nu_{\text{nb}}^{(i)}\}$.

t(`ncomp=1, dtype=c_long`) stores the number of classified bonds of particle i.

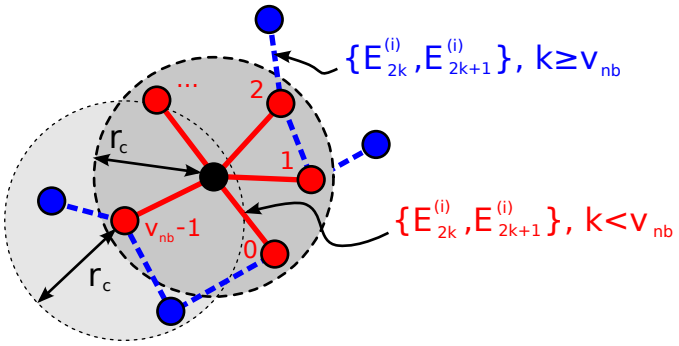

Figure 2-5: *Local bonds used for CNA construction*

Using those `ParticleDats`, for each particle the list representation $E^{(i)}$ of the set $\mathcal{E}_d^{(i)} \cup \overline{\mathcal{E}}^{(i)}$ can now be calculated with two Local Particle Pair Loops: the first loop, shown in Algorithm 4, calculates the first $2\nu_{\mathrm{nb}}^{(i)}$ entries of $E^{(i)}$ by inspecting the direct neighbours of each particle. Based on this, the second loop in algorithm 5 adds the remaining $2(\nu_{\mathrm{b}}^{(i)} - \nu_{\mathrm{nb}}^{(i)})$ entries, i.e. the blue (dashed) lines in Figure 2-5. The final Particle Pair Loop in algorithm 6 then uses the information stored in $E^{(i)}$ and $E^{(j)}$ to extract the tuple $(n_{\mathrm{nb}}, n_{\mathrm{b}}, n_{\mathrm{lcb}})$.

Algorithm 4: CNA Local Particle Pair Loop I: Calculate direct bonds for each particle.

Data: $\vec{r}^{(i)}$ [`READ`], $G^{(i)}$ [`READ`].
Result: $\nu_{\mathrm{nb}}^{(i)}$ [`INC_ZERO`], $\nu_{\mathrm{b}}^{(i)}$ [`INC_ZERO`], $E^{(i)}$ [`WRITE`]
for *pairs* (i, j) **do**
  if $|\vec{r}^{(i)} - \vec{r}^{(j)}| < r_c$ **then**
    $(E_{2\nu_{\mathrm{b}}}^{(i)}, E_{2\nu_{\mathrm{b}}+1}^{(i)}) = (G^{(i)}, G^{(j)})$
    $\nu_{\mathrm{b}}^{(i)} \leftarrow \nu_{\mathrm{b}}^{(i)} + 1$
    $\nu_{\mathrm{nb}}^{(i)} \leftarrow \nu_{\mathrm{nb}}^{(i)} + 1$
  end
end

Algorithm 5: CNA Local Particle Pair Loop II: Calculate all other bonds in the local environment.

Data: $\vec{r}^{(i)}$ [`READ`], $G^{(i)}$ [`READ`], $\nu_{\mathrm{nb}}^{(i)}$ [`READ`].
Result: $\nu_{\mathrm{b}}^{(i)}$ [`INC`], $E^{(i)}$ [`RW`]
for *pairs* (i, j) **do**
  if $|\vec{r}^{(i)} - \vec{r}^{(j)}| < r_c$ **then**
    for $k = 0, \ldots, \nu_{\mathrm{nb}}^{(j)} - 1$ **do**
      if $E_{2k+1}^{(j)} \neq G^{(i)}$ **then**
        $(E_{2\nu_{\mathrm{b}}}^{(i)}, E_{2\nu_{\mathrm{b}}+1}^{(i)}) = (E_{2k}^{(j)}, E_{2k+1}^{(j)})$
        $\nu_{\mathrm{b}}^{(i)} \leftarrow \nu_{\mathrm{b}}^{(i)} + 1$
      end
    end
  end
end

Algorithm 6: CNA Local Particle Pair Loop III: Calculate number of common neighbours $n_{\mathrm{nb}}^{(i)}$, number of bonds $n_{\mathrm{b}}^{(i)}$ between those common neighbours and the largest clustersize $n_{\mathrm{lcb}}^{(i)}$.

Data: $\vec{r}^{(i)}$ [`READ`], $\nu_{\mathrm{nb}}^{(i)}$ [`READ`], $\nu_{\mathrm{b}}^{(i)}$ [`READ`], $E^{(i)}$ [`READ`].
Result: $T^{(i)}$ [`WRITE`], $t^{(i)}$ [`INC_ZERO`].
for *pairs* (i, j) **do**
  if $|\vec{r}^{(i)} - \vec{r}^{(j)}| < r_c$ **then**
    Set $\mathcal{C}$ of common neighbours:
    $\mathcal{C} \leftarrow \{v : \exists k < \nu_{\mathrm{nb}}^{(i)}, \ell < \nu_{\mathrm{nb}}^{(j)}, v = E_{2k+1}^{(i)} = E_{2\ell+1}^{(j)}\}$
    Construct set $\mathcal{E}$ of common neighbour bonds:
    $\mathcal{E} \leftarrow \{\}$
    for $k = \nu_{\mathrm{nb}}^{(i)}, \ldots, \nu_{\mathrm{b}}^{(i)} - 1$ **do**
      if $E_{2k}^{(i)} \in \mathcal{C}$ *and* $E_{2k+1}^{(i)} \in \mathcal{C}$ **then**
        $(v, w) = (E_{2k}^{(i)}, E_{2k+1}^{(i)})$
        if $w > v$ **then**
          swap $v \leftrightarrow w$
        end
        if $(v, w) \neq \mathcal{E}$ **then**
          $\mathcal{E} \leftarrow \mathcal{E} \cup (v, w)$
        end
      end
    end
    $T_{3t^{(i)}}^{(i)} \leftarrow |\mathcal{C}|$
    $T_{3t^{(i)}+1}^{(i)} \leftarrow |\mathcal{E}|$
    Calculate largest cluster size, see Algorithm 25:
    $T_{3t^{(i)}+2}^{(i)} \leftarrow$ `maxClustersize`$(\mathcal{E})$
    $t^{(i)} \leftarrow t^{(i)} + 1$
  end
end

CHAPTER 3

CODE GENERATION OF MODERN PARALLEL MD ALGORITHMS

In this chapter we discuss cell based methods, these are a major component in our parallel decomposition approach and our implementation of *Particle Loops* and *Local Particle Pair Loops*. Furthermore, we provide an overview of our code generation process and present results from parallel simulations and analysis techniques.

3.1 Modern Parallel MD Algorithms

3.1.1 Cell Based Methods

Execution of a *Local Particle Pair Loop* can be constructed from two components: firstly, find all pairs of particles (i, j) with positions $(\vec{r}_i, \vec{r}_j)$ such that $|\vec{r}_i - \vec{r}_j| < r_c$, secondly, given pairs of particles execute the provided kernel on each particle pair. We shall discuss existing methods where these two components are combined into one operation and methods where the two stages are treated separately.

All of the methods we describe to identify pairs of particles are known as cell based methods. Cell based methods decompose the simulation domain into so called cells which have side lengths greater than the exclusion radius, r_c. We discuss how cell based domain decompositions, firstly, facilitate the efficient discovery of particle pairs and, secondly, improve the efficiency of MPI communication.

Throughout this section we shall assume that the simulation domain is a cuboid with extents L_x, L_y, L_z and that the cells are also cuboid shaped. Following the approach of Rapaport [17] the simulation domain is decomposed into a cell grid of integer dimensions G_x, G_y, G_z where each cell has side lengths w_x, w_y, w_z and N_c is the total number of cells. Cells are indexed lexicographically from zero with the map $c_i = \mathcal{C}(c_i^{(x)}, c_i^{(y)}, c_i^{(z)}) = c_i^{(x)} + c_i^{(y)} G_x + c_i^{(z)} (G_x G_y)$. Where $c_i^{(x)} = 0, \ldots, G_x - 1$, $c_i^{(y)} = 0, \ldots, G_y - 1$ and $c_i^{(z)} = 0, \ldots, G_z - 1$. Given a cell grid we use Algorithm 7 to determine the cell c_i that contains

a particle i.

Algorithm 7: Method to determine containing cell c_i of particle i.

Data: Particle position $\vec{r}_i$, domain extent $\vec{l} = (L_x, L_y, L_z)$, cell array G_x, G_y, G_z and cell edge lengths w_x, w_y, w_z.
Result: c_i: cell containing particle i.

$$\vec{r'}_i = \vec{r}_i + \frac{1}{2}\vec{l} \tag{3.1}$$

$$c_i^{(x)} = \left\lfloor \frac{\vec{r'}_i^{(x)}}{w_x} \right\rfloor, c_i^{(y)} = \left\lfloor \frac{\vec{r'}_i^{(y)}}{w_y} \right\rfloor, c_i^{(z)} = \left\lfloor \frac{\vec{r'}_i^{(z)}}{w_z} \right\rfloor \tag{3.2}$$

$$c_i = c_i^{(x)} + c_i^{(y)} G_x + c_i^{(z)} (G_x G_y) \tag{3.3}$$

Efficiently computing the cell c_i containing a given particle with index i is not typically problematic even on novel hardware architectures such as GPUs. However, cell based methods require a map from cell index to the indices of particles contained within the cell, which should be efficient to construct and evaluate. In a slight abuse of terminology, a cell to particle map $\mathcal{Q}$ is defined such that $\mathcal{Q}(m)$ is the set of particles i such that $c_i = m$. Most often an algorithm to build and evaluate this map which is efficient on one hardware architecture will not be efficient on another. We shall describe two methods to construct the cell to particle map, one which is efficient on CPU architectures in a MPI only setting, the second is designed to be efficient on GPU architectures and can also be applied in a CPU shared memory model.

Assuming the map from cells to particles $\mathcal{Q}$ exists we can identify all pairs of particles (i, j) such that $|\vec{r}_i - \vec{r}_j| < r_c$ without inspecting all particle pairs. The approach will propose pairs of particles such that $|\vec{r}_i - \vec{r}_j| \geq r_c$ and these are excluded at some later stage, we describe methods to efficiently exclude these particles by using neighbour list techniques. Consider a particle i in a cell $c_i = c_i^{(x)} + c_i^{(y)} G_x + c_i^{(z)} (G_x G_y)$ neighbouring particles j are contained in cells $c_j = c_i + d$ where $d = d^{(x)} + d^{(y)} G_x + d^{(z)} G_x G_y$ for $(d^{(x)}, d^{(y)}, d^{(z)}) \in \{\vec{d} \in \mathbb{Z}^3 : |\vec{d}|_\infty = 1\} \cup \{0\}$, i.e. the cell c_i itself and the surrounding 26 cells. Here we have assumed that the extent of the cells are at least the interaction cutoff r_c, in principle this constraint can be relaxed if more neighbouring cells are considered. Figure 3-1 demonstrates the cells that should be inspected for potential neighbours of a selected particles.

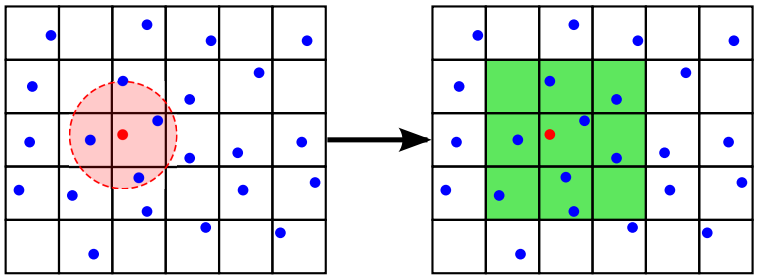

Figure 3-1: *Left: Circle of radius r_c around a selected particle. Right: Corresponding cells (in 2D) that contain all potential neighbours within a radius r_c.*

3.1.2 Parallel Decomposition

We apply a domain decomposition approach to distribute computational work across MPI ranks. The simulation domain is subdivided into sub-domains based on the number of MPI ranks available, each MPI rank owns a sub-domain and owns the particles that are contained within the sub-domain. For example, in Figure 3-2 a domain is decomposed across 4 MPI ranks. The sub-domains assigned to each MPI rank are identical in shape and hence if the distribution of particle positions within the simulation domain is uniform then each MPI rank is on average assigned the same computational work. Plimpton, S. et al [63] discuss two other possible work decomposition methods, both permanently assign particles to MPI ranks at the beginning of the simulation, this approach results in additional communication overhead between MPI ranks in comparison to a domain decomposition approach.

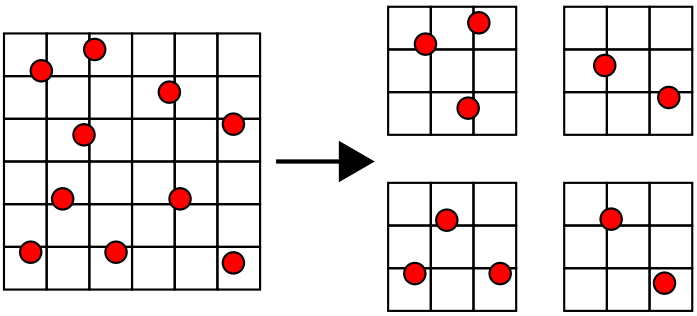

Figure 3-2: *Decomposing a domain into four sub-domains.*

In our approach the domain decomposition across MPI processes is applied before the cell decomposition, this ordering assigns an equal volume of domain to each MPI rank. If the order is reversed and cell decomposition occurs before domain decomposition then implementation issues arise when the number of cells is not an integer multiple of the number of MPI ranks in each dimension. In this scenario, which is highly likely, either sub-domain boundaries are along cell boundaries or sub-domain boundaries divide cells in some way. The first case where sub-domain boundaries align with cell boundaries causes load imbalance for smaller cell counts the alternative method increases the complexity of "bookkeeping" operations and MPI communications between MPI ranks.

It is expected that particles will move throughout the simulation and hence particles will regularly move between sub-domains. When this happens the ownership of the particle is transferred from the source to the destination MPI rank. The `State` class automatically generates code to move all properties of transferred particles which are stored in `ParticleDat` objects between sub-domains when required.

3.1.3 Halo Exchange

The parallel efficiency of a *Local Particle Pair Loop* is highly dependent on the particular MPI communication pattern between sub-domains. This communication pattern is often referred to as a "halo exchange" and is common in parallel scientific code that applies domain decomposition. The halo exchange can be thought of as a bridge between sub-domains, in our case particles owned by a sub-domain will interact with particles owned by a neighbouring sub-domain. Hence any data required to compute interactions, e.g. particle positions, is copied from the owning MPI rank in the halo exchange. Copying data between MPI ranks is very slow in comparison to CPU computation and hence we minimise the amount of data copied and the frequency with which it is copied by inspecting access descriptors. We refer to the region of the sub-domain that contains data duplicated from neighbouring sub-domains as the halo region.

The frequency of halo exchange operations is minimised through access descriptors and "dirty" flags on `ParticleDat` instances. When a halo exchange is performed on a particle property the halo regions remain valid until the data is marked as dirty. For example, if two *Particle Pair Loops* are launched that only read particle positions then only at most one halo exchange must be performed.

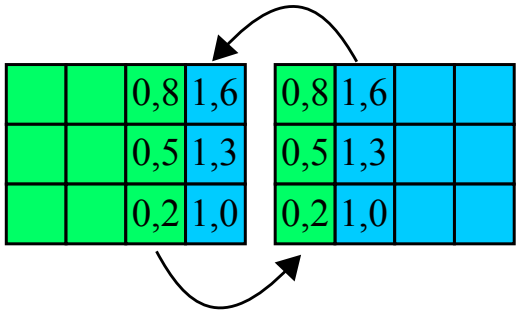

Figure 3-3: *Halo exchange that must occur between two horizontally adjacent sub-domains.*

The volume of transferred data is minimised by considering the cell structure imposed on each sub-domain. In each dimension the cell extents are at least the interaction cutoff r_c by construction, from a sub-domain only the outer shell of cells can ever be required by a neighbouring sub-domain. Figure 3-3 illustrates the halo exchange pattern between two sub-domains. Hence the cell to particle map facilitates efficient inter-process communication in addition to efficient *Local Particle Pair Loop* execution. From an implementation perspective, the halo exchange of particle data is more complex than in a grid based sci-

entific code where the number of exchanged elements can be deduced from the underlying grid and does not change with time. In a MD simulation cells contain numbers of particles that vary between steps and between cells, these need to be efficiently packed and communicated.

Each sub-domain is surrounded by 26 neighbouring sub-domains, one neighbour for each face, edge and vertex of the sub-domain. For the fully periodic case, the neighbours of a sub-domain on the boundary exist over the boundary. In the naive approach each sub-domain separately packs and exchanges data with each of its 26 neighbours, we implement the process described by Plimpton [63] that performs all required data movement in 6 exchanges.

A 2D illustration of this 6-exchange pattern is given in Figure 3-4. The 3 pairs of opposite faces of the sub-domain are labelled as (north, south), (east, west) and (up, down). First, all sub-domains halo exchange in the north and south directions, i.e for the north exchange each process packs and sends the particle data required by the sub-domain to the north and receives from the sub-domain to the south, this is then repeated in the opposite direction. All (north, south) communication is contention free and both directions can be performed simultaneously with a MPI `Sendrecv` operation. The process is now repeated in the (east, west) directions with the exception that each sub-domain packs the particles it owns in boundary cells and includes particles which were received in the (north, south) exchange. In the final stage each sub-domain packs and sends particle data it owns and particle data that was received in the previous two exchanges to perform the (up, down) exchanges.

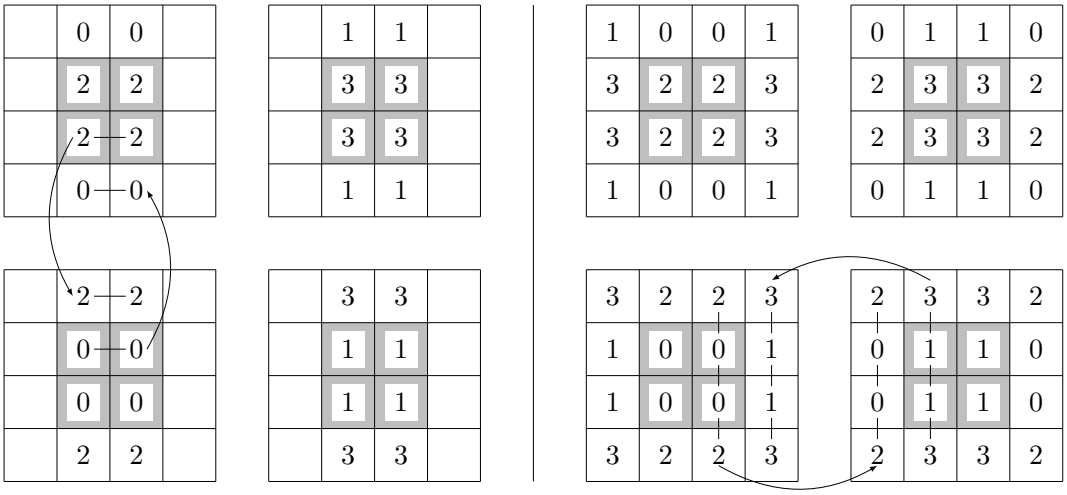

Figure 3-4: *2D version of fully periodic halo exchange pattern. 4 sub-domains, each own a 2×2 grid of cells shown in grey. Numbers in cells indicate the sub-domain that owns data in the cell. Left: (north, south) exchange between sub-domains 0 and 2 indicated by arrows. Right: (east, west) exchange indicated between sub-domains 0 and 1. Note the second exchange (right) includes the data from the first (left) exchange.*

3.1.4 Cell To Particle Maps

We describe two cell to particle map approaches, one which is typically efficient on CPU architectures and one which is efficient on GPU architectures. These cell to particle maps are utilised as a first stage in all *Local Particle Pair Loops* we implement and are used to implement efficient halo exchanges on both architectures.

CPU

For N particles in a sub-domain of N_c cells the cell list method proposed by Rapaport [17] constructs a forward linked list in an array $q \in \mathbb{Z}^{N+N_c}$. The linked list approach has known and constant memory requirements and typically requires less memory for storage than alternative methods. The manner in which the list is built is not well suited for highly threaded architectures, such as GPUs, where atomic operations are relatively expensive. The linked list is implemented as an array q, for each index $i \in \{0, N-1\}$ the "current" particle index is i and the next particle index is q_i. An entry of $q_i = -1$ indicates that i is the last particle index in the list. Elements N to $N + N_c - 1$ store the index of the first particle in the list for each cell. An algorithm to construct the array q is given in Algorithm 8 and an example illustration is given in Figure 3-5.

Algorithm 8: Construction of linked list cell list.

Data: Particle positions $\vec{r}_i,\ i = 0, \ldots, N-1$ and N_c cells.
Result: Linked list q

Initialise list by resetting the lookup part of the list.
for $c = 0, \ldots, N_c - 1$ **do**
  $q_{N+c} = -1$
end
Populate the list with particle indices.
for $i = 0, \ldots, N-1$ **do**
  get cell c_i containing particle i from Algorithm 7
  $q_i = q_{N+c_i}$
  $q_{N+c_i} = i$
end

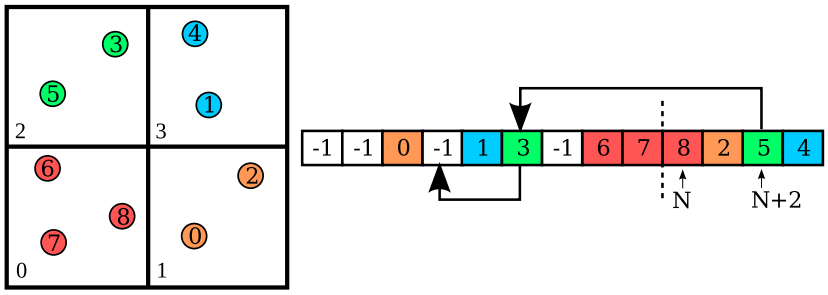

Figure 3-5: *Example of a cell list (right) constructed from a sub-domain with 4 cells (left). The arrows follow the path traced to retrieve the indices of particles in cell 2.*

GPU

If the cell list algorithm is implemented on a highly threaded shared memory architecture, such as a GPU, then the update stages of q in Algorithm 8 will exhibit enough write contention to render the algorithm inefficient. Rapaport [17] describes a cell occupancy matrix $H_{c,l}$ where row c sequentially stores the indices of particles in cell c. The matrix is constructed by first looping over particles to determine which cell they reside in and to determine which layer in the cell they are in, layers are assigned to give an order to the particles in a cell. The layer a particle is given determines which column in the cell occupancy matrix the particle index should be placed. Hence if particle i is in cell c_i on layer l_i then $H_{c_i,l_i} = i$. Algorithm 9 provides an outline of this method and a 2D illustration is provided in Figure 3-6.

Algorithm 9: Assigning layers to particles and determining cell occupancy counts. Rapaport [17] Section 3.4.

Data: Particle positions $\vec{r}_i,\ i = 0, \ldots, N-1$ and N_c cells.

Result: Occupancy matrix H and cell occupancy counters k.

Reset cell counters:

for $c = 0$ **to** $N_c - 1$ **do**
  $k_c = 0$
end

for $i = 0$ **to** $N - 1$ **do**
  get cell c_i containing particle i from Algorithm 7
  $k_{c_i} = k_{c_i} + 1$ (atomic increment)
  $l_i = k_{c_i}$
end

Populate cell occupancy matrix:

for $i = 0$ **to** $N - 1$ **do**
  $H_{c_i,l_i} = i$
end

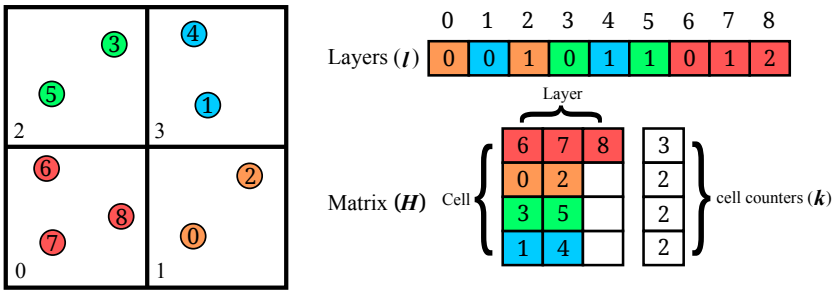

Figure 3-6: *Example of H, l and k for a simple 4 cell sub-domain.*

From a practical perspective, Algorithm 9 is broken into two sections, in the first particle layers and cell occupancies are computed and in the second stage storage is allocated for H and subsequently H is populated. If the particles are not uniformly distributed in the domain and are instead clustered in a region then the occupancy matrix approach will require significantly more memory than the cell list approach. However, the sequential arrangement of particle indices within the occupancy matrix is much more favorable on hardware architectures where random access to memory is expensive than the cell list approach.

3.1.5 Finding And Storing Pairs Of Particles

We now describe how pairs of particles are identified by using a constructed cell to particles map.

Cell By Cell

In cell by cell methods, interacting particles are identified by simply using the cell to particle map to list the indices of particles in the cells surrounding a chosen particle i. In Algorithm 10 we provide an overview of a cell by cell approach by Rapaport [17] that loops over all cells in the sub-domain and for each cell considers all pairs of particles formed between the initial cell and all neighbouring cells.

Algorithm 10: Propose pairs of particles by considering pairs of cells.

Data: N_c cells, cell list q and particle positions $\vec{r}$
Result: Kernel launch over all required pairs.
for $c = 0, \ldots, N_c - 1$ **do**
  for $k = 0, \ldots, 26$ *neighbouring cells* **do**
    $c' = c + k^{\text{th}}$ offset to neighbouring cell
    $i = q_{N+c}$
    while $i > -1$ **do**
      $j = q_{N+c'}$
      while $j > -1$ **do**
        if $|\vec{r}_i - \vec{r}_j| < r_c$ *and* $i \neq j$ **then**
          Execute kernel, e.g. compute force between particles i and j.
        end
        $j = q_j$
      end
      $i = q_i$
    end
  end
end

Although we describe this cell by cell approach using a cell list, such as the one constructed in Algorithm 8, the method can be applied using any cell to particle map. This method makes no attempt to store particle pairs and in practice exposes a significant number of particle pairs which are very well separated and for these pairs the kernel will be executed unnecessarily. To see this inefficiency, consider a system with particle density ρ where the cell extent and interaction cutoff is r_c. If one particle is considered, this particle is paired with all other particles within the neighbouring 27 cells containing approximately $27\rho r_c^3$ particles. However, the sphere of radius r_c around the chosen particle contains $\frac{4}{3}\pi r_c^3 \rho \approx 4\rho r_c^3$ particles. Hence if we could only propose neighbours in the sphere of radius r_c as opposed to the 27 cells of extent r_c then the pairwise kernel will be wastefully executed for a fewer number of well separated particles by a factor of $81/(4\pi) \approx 6.4$, Figure 3-7 illustrates this ratio in 2D.

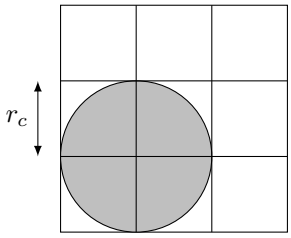

Figure 3-7: *2D comparison between circle of radius r_c and cells inspected for neighbours.*

Cell by cell methods can be implemented in a manner that very efficiently utilises

SIMD floating point units common in modern CPUs. An efficient technique loads all required properties from all particles in the cells c_i and c_j into temporary arrays in a gather operation. This gather operation collects particle data contiguously in the temporary arrays which is highly favourable in comparison to the "global" store of particle data where data access is most likely to be random. Kernel execution is performed ideally using SIMD instructions using the temporary arrays and after the kernel is launched on all particle pairs written values are copied back into the global store.

The efficiency of the cell by cell approach can potentially be increased by reducing the extent of the cells to a fraction of the cutoff radius r_c and increasing the number of neighbouring cells that are considered. The smaller cells form a better approximation of the sphere of interaction and hence less particle pairs are proposed that are very well separated. Alternatively, the cell by cell technique is used to discover pairs of particles that could be within the interaction radius r_c of each other, these pairs are then stored in a data structure known as a neighbour list.

More formally, a neighbour list is a list of pairs of particles (i, j) such that $|\vec{r}_i - \vec{r}_j| < r_n$ where r_n is a cutoff radius and $\vec{r}_i$ and $\vec{r}_j$ are particle positions. Typically, $r_n := r_c + \delta$ where r_c is the cutoff of a particular interaction and δ is a buffer region. The buffer region is chosen such that $\delta = v_{\max} 2N_s$ where $v_{\max}$ is the maximum particle velocity in any direction and $N_s \in \mathbb{N}$ is a number of time step iterations for which we want to guarantee that no particle has crossed half the buffer region. By adding a buffer region a neighbour list can be constructed and reused for N_s iterations. The construction of neighbour lists is often sufficiently expensive, especially on GPU architectures, that neighbour list approaches are not efficient if the lists are not reused. We now describe two existing neighbour list approaches, one that is efficient on single-threaded CPU architectures and one that is efficient in highly threaded shared memory architectures such as GPUs and multicore CPUs.

Sequential Neighbour Lists

This approach, described by Rapaport [17], uses a cell list to construct a neighbour list in a sequential manner that is efficient in terms of memory requirements. For each particle i neighbouring particle indices j are stored sequentially in an array $\vec{b}$ such that all the neighbours of i form a contiguous block in $\vec{b}$. We refer to this method as "sequential" as the indices of neighbours of a particle i are stored adjacent to the neighbours of particle $i + 1$. For N particles an auxiliary array $\vec{s} \in \mathbb{N}^{N+1}$ stores the location of the neighbours of each particle in $\vec{b}$ by setting $\vec{s}_i$ to be the index of the first neighbour of i in $\vec{b}$, the last element of $\vec{s}$ is assigned the appropriate terminating value.

Algorithm 11: Construction of sequential neighbour list.

Data: N particles, cell list q, particle positions $\vec{r}$ and cutoff radius r_n
Result: neighbour list $\vec{b}$, starting points $\vec{s}$

$m = -1$
for $i = 0, \ldots, N-1$ **do**
  Determine containing cell: c_i.
  $\vec{s}_i = m + 1$
  for $k = 0, \ldots, 26$ *neighbouring cells* **do**
    $c' = c_i + k^{\text{th}}$ offset
    $j = q_{N+c'}$
    Loop over potential neighbours in cell c'.
    while $j > -1$ **do**
      if $i \neq j$ *and* $|\vec{r}_i - \vec{r}_j| < r_n$ **then**
        $m = m + 1$
        $\vec{b}_m = j$
      end
      $j = q_j$
    end
  end
end
Terminate the list of neighbours of particle $N-1$.
$\vec{s}_N = m + 1$

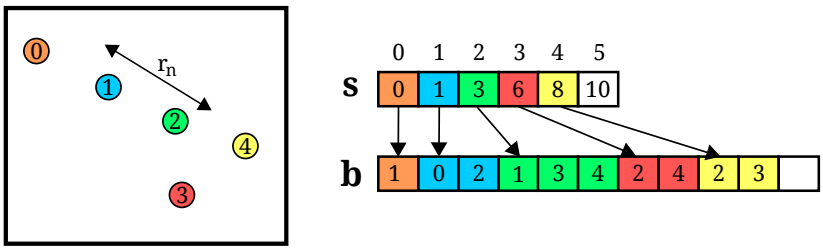

Figure 3-8: *Example of the neighbour list and associated starting points.*

An overview of the construction of the neighbour list is provided in Algorithm 11 and an illustration of the approach is given in Figure 3-8. This data structure is not data parallel due to the adjacent arrangement of particle neighbours, this limitation is non-trivial to overcome with atomic operations and hence the algorithm is not suitable for shared memory architectures. Given a sequential neighbour list $(\vec{s}, \vec{b})$ a kernel can be executed over pairs of particles using Algorithm 12.

Algorithm 12: Pairwise kernel execution using a sequential neighbour list.

```
Data: neighbour list b⃗, starting points s⃗, particle positions r⃗ and cutoff radius r_c
for i = 0, ..., N − 1 do
    for k = s⃗_i, ..., s⃗_{i+1} − 1 do
        j = b⃗_k
        if |r⃗_i − r⃗_j| < r_c then
            Execute kernel, e.g. compute force between particles i and j.
        end
    end
end
```

Unlike the cell by cell approach the sequential neighbour list loses the cell structure from which it was constructed. The inner most loop over particle neighbours j has the potential to access memory in a particularly inefficient manner, if the particle data of neighbours j is scattered across the global particle data store then access to the particle data of neighbours is essentially random.

A random access pattern can be partially mitigated by periodically reordering particle data in memory such that particles which are contained within the same cell are adjacent in memory. Ideally the order in which particle data is stored should be the same as the order particle indices are stored in the cell to particle map, hence in neighbour list construction particle indices are identified and stored in the same order as particle data. If this reordering is performed then the probability that the data of neighbour j is adjacent to the data of neighbour $j+1$ is increased. Furthermore, a cell based reordering potentially decreases the CPU cache miss rate as the neighbours of particle i are likely to overlap with the neighbours of particle $i + 1$.

Matrix Neighbour Lists

The sequential neighbour list is not data parallel and hence cannot efficiently be implemented in highly threaded shared memory environments, to address this problem we describe the neighbour matrix approach by Rapaport [17]. This is a method suitable for implementation on GPU architectures and hence we focus on GPU specific details, the method is also efficient in a shared memory model with minor modifications.

For N particles a data parallel neighbour list is created using a matrix with N columns and some reasonable number of rows corresponding to a maximum number of neighbours per particle. The matrix $W_{m,i}$ stores the neighbours of particle i in column i with all m^{th} neighbours on the m^{th} row. All neighbour indices of a particle i are identified and stored by a single GPU thread to avoid write contention.

This particular layout ensures that the indices of all m^{th} neighbours are sequential in memory. GPU threads are assigned to particles such that thread i loops over all neighbours of particle i, the threads in a block of GPU threads are expected to require the

m^{th} neighbour indices simultaneously. With this data layout, contiguous thread indices are accessing contiguous entries in the neighbour matrix simultaneously, which is the most efficient memory access pattern on modern GPU architectures. Algorithm 13 constructs a neighbour matrix H and auxiliary array t that holds the number of neighbours of each particle from a cell occupancy matrix H.

Algorithm 13: Construction of matrix neighbour list based on Rapaport [17] Section 3.4.

Data: N particles, cell occupancy matrix H, cutoff r_n and particle positions $\vec{r}$
Result: Neighbour matrix W and neighbour count array t
for $i = 0$ **to** $N - 1$ **do**
  Determine containing cell: c_i.
  $m = 0$
  for $k = 0$ **to** 26 *neighbouring cells* **do**
    $c' = c_i + k^{\text{th}}$ offset
    for $l = 0$ **to** $k_{c'} - 1$ **do**
      $j = H_{c',l}$
      if $i \neq j$ *and* $|\mathbf{r}_i - \mathbf{r}_j| < r_n$ **then**
        $W_{m,i} = j$
        $m = m + 1$
      end
    end
  end
  $t_i = m$
end

Pairs of particles are exposed to a pairwise kernel using Algorithm 14, this looping pattern is extremely similar to the sequential neighbour list method. As with the sequential neighbour list method the neighbour matrix potentially accesses particle data in a highly inefficient manner if particle data is arranged in a global store in a unstructured arrangement. On GPU architectures rearrangement of particle data to group the data by sub-domain cells is highly recommended for efficient data access.

Algorithm 14: Interaction using matrix neighbour list.

Data: neighbour matrix W, neighbour counts t, particle positions $\vec{r}$ and cutoff radius r_c

for $i = 0, \ldots, N-1$ **do**
  for $k = 0$ **to** $t_i - 1$ **do**
    $j = W_{i,k}$
    if $|\vec{r}_i - \vec{r}_j| < r_c$ **then**
      Execute kernel, e.g. compute force between particles i and j.
    end
  end
end

3.1.6 Neighbour List Rebuilding

We intend to build a neighbour list using a cutoff radius $r_n = r_c + \delta$ where $\delta = v_{\max} 2 N_s$. By padding the radius used to perform cell decomposition and neighbour list construction we allow particles to move small distances without invalidating the cell to particle map and neighbour lists, however, these will both need to be reconstructed periodically to remain valid.

An implementation of a neighbour list algorithm requires the number of steps N_s and a mechanism to compute the maximum velocity of any particle in any direction $v_{\max}$. We provide the `IntegratorRange` class which allows time step based methods to be implemented in a Python `range` like loop as in Listing 3.1. Use of this class allows neighbour list based pair looping methods to be used by the user without explicit calls to rebuild cell to particle maps and neighbour lists.

Listing 3.1: *Example use of* `IntegratorRange` *called with:* `Ni` *number of iterations, timestep size* `dt`*, velocities* `v`*, list reuse count* `Ns` *and shell thickness* `delta` $= r_n - r_c$.

```
for i in IntegratorRange(Ni, dt, A.v, Ns, delta):
    particle_loop_1.execute()
    force_calculation.execute()
    particle_loop_2.execute()
```

3.2 Code Generation

The project is primary implemented in Python [66] which is a high-level, dynamically typed and object oriented programming language. Python allows for the easy expression of algorithms in a high-level manner, for example, time stepping. Python interpreters exist for most modern operating systems (GNU Linux variants) used on HPC hardware allowing framework code and user code to be ported between platforms without modification. The

Python community has developed a large collection of libraries for scientific and non-scientific purposes which usually can be easily imported and used under open source licenses.

Python is considered as a highly productive language in terms of programmer time due to its flexibility, but the language is typically not efficient in terms of CPU usage. Firstly, Python is an interpreted language with typically no optimisation before execution, secondly, Python is dynamically typed meaning the interpreter must check the type of objects each time they are parsed. Finally, Python has poor native support for multi-core programming, the global interpreter lock prevents efficient shared memory threading techniques as only one thread may interpret source code at a time. Basic multiprocessing support does natively exist within Python, we use the `mpi4py` [15] package which provides Python bindings to an underlying MPI library.

The C programming language is a suitable choice to generate code in for multiple reasons, the primary reason is that well written C code in combination with a modern C compiler can be highly efficient. C compilers typically have multiple optimisation stages that, with a suitable input, produce efficient machine code. Furthermore, C shared libraries have a well defined interface that allows Python code to load and use functions in shared libraries in a robust manner. As C is an established language compilers are available on all major platforms and it is reasonable to assume that future hardware will have support for the C language. We combine Python and C to leverage the flexibility of Python with the efficiency possible from C.

We use code generation as it is highly flexible in comparison to alternative methods that combine Python and C. With careful software engineering it is reasonably straightforward to implement computationally expensive operations in C by hand as a library which can be used from Python. In this static approach, new functionality must be added by writing C code, which is a process we explicitly wish to minimise as the technical nature of the process may not be in the skill set of the domain specialist. By using our code generation framework a user benefits from efficient C libraries without hand writing the library. Furthermore, the separation of concerns approach allows computational scientists to optimise the code generation system and target new hardware architectures as a continuous process.

Code generation enables high-level optimisations which are typically not possible with static C libraries. For example, loop fusion is an optimisation where initially separate loops that access common data are identified and combined, this removes duplicate memory access. If kernels are combined there is a greater probability that a C compiler or a code generation framework can identify and perform common sub-expressions elimination, this is an optimisation where duplicated expressions are computed once and the resulting value stored to remove duplicated work. Furthermore, code generation allows efficient C code to be constructed and executed by non-technical users. In the Firedrake [67] project users formulate input as symbolic expressions, the project generates and executes optimised C code in parallel in a manner that is invisible to the user.

In our framework we minimise inter-process communication by avoiding unnecessary communication and generating efficient packing and unpacking functions. We provide efficient implementations of our two looping types for two hardware architectures by generating code from kernels and access descriptors with a relatively small code base. We now describe the code generation system responsible for executing the *Particle Loop* and *Local Particle Pair Loop* looping mechanisms.

Assumptions And Definitions

We assume that a system of N particles is decomposed across P MPI ranks such that MPI rank k owns $N^{(k)}$ particles, ideally $N^{(k)} = N/P$ for all k. Furthermore, we assume the loop L is defined by

$$L = \left(K, D^{(p)}, A^{(p)}, D^{(s)}, A^{(s)}\right), \tag{3.4}$$

where K is a `Kernel` instance containing a C string written with the syntax described in Section 2.3.1. $D^{(p)}$ is a collection of $m^{(p)}$ `ParticleDat` instances with access descriptors $A^{(p)}$ such that

$$D^{(p)} = \{d_0^{(p)}, \ldots, d_{m^{(p)}}^{(p)}\}, \tag{3.5}$$

$$A^{(p)} = \{a_0^{(p)}, \ldots, a_{m^{(p)}}^{(p)}\}. \tag{3.6}$$

Furthermore, $D^{(s)}$ is a collection of $m^{(s)}$ `ScalarArray` or `GlobalArray` instances with access descriptors $A^{(s)}$ such that

$$D^{(s)} = \{d_0^{(s)}, \ldots, d_{m^{(s)}}^{(s)}\}, \tag{3.7}$$

$$A^{(s)} = \{a_0^{(s)}, \ldots, a_{m^{(s)}}^{(s)}\}. \tag{3.8}$$

We now give a technical overview of our code generation method for *Particle Loops* and *Local Particle Pair Loops*. For both looping types our implementation considers each data structure instance in turn, for each `ParticleDat`, `ScalarArray` and `GlobalArray` C code is generated that provides the data access required by the corresponding access descriptor. The generation process produces C code such as: function declarations, structure declarations, indirection indices and loops.

3.2.1 Particle Loop

A *Particle Loop* implementation is required to execute the kernel K on each particle as described in Definition 2.1. Fundamentally, this operation is a single loop over all particles where properties $\pi^{(i)}$ of particle i are accessed at most once, hence access to particle data is a data parallel operation. Access to global properties must be performed in a manner that is independent of the order particles are considered, i.e. operations involving global properties are associative. Here we shall describe the process to generate C code that implements a *Particle Loop* for CPU architectures, in Appendix A.8.1 we describe the

corresponding process for GPU architectures.

We describe the code generation process to implement L for a *Particle Loop* without any shared memory techniques, however, our framework implementation will generate C code using shared memory parallelism with OpenMP. This description produces a valid C implementation for L and describes the process all our code generation methods apply to create a shared library that contains an externally callable function. This function is called by the Python implementation to execute the loop. The general structure of this kind of library is given in Listing 3.2.

Listing 3.2: *Overview of Particle Loop C*

```
// Structs generated per ParticleDat
<generated_structs>

// kernel source wrapped in a function
// ParticleDats are passed using above generated structs
// ScalarArray and GlobalArrays are passed as pointers
inline void k_<kernel_name>(<kernel_parameter_list>){
    <kernel_source>
}

// Externally available function to be called from Python
void <kernel_name>_wrapper(const int _N_LOCAL,
   <data_structure_pointers>){

    // loop over all owned particles
    for(int _i=0 ; _i<_N_LOCAL ; _i++){

        // create instances of generated structs
        // for each ParticleDat
        <kernel_args_creation>

        // call kernel
        k_<kernel_name>(<kernel_call>);

    }
}
```

In Listing 3.2 `<kernel_name>` is immediately substituted for the name the user gave to the kernel, this name is arbitrary and is mentioned for completeness. All other substitution locations in the code are filled with C code generated from the passed data structures and accompanying access descriptors. Code is generated for the `ParticleDat` instances as in Algorithm 15 and for `ScalarArray` and `GlobalArray` instances in Algorithm 16.

For each `ParticleDat` instance we create a C structure declaration to use as an interface between the user written kernel and the generated looping code. For each particle index an instance of this structure is created and passed as an argument to a generated

function that contains the kernel. Typically, with optimisation enabled, modern compilers will inline the kernel function as requested and perform the indirection described by the `ParticleDat` structure without actually creating instances of the structure.

Algorithm 15: Particle Loop code generation for `ParticleDats`

Data: `ParticleDat` instances $D^{(p)}$ and access descriptors $A^{(p)}$

Result: `<generated_structs>`, `<kernel_args_declaration>`, `<data_structure_pointers>`, `<kernel_args_creation>` and `<kernel_call>`

for $m \in \{0, \ldots, m^{(p)}\}$ **do**

Construct identifier `sym` to use for temporary variables (usually the symbol used in the kernel)

Identify underlying data type `dtype` from $d_m^{(p)}$

Identify number of components `ncomp` from $d_m^{(p)}$

Determine if the `const` qualifier is valid from $a_m^{(p)}$

(1) Create `struct` for `<generated_structs>`:
`typedef struct {dtype (const) *i;} _sym_t;`

(2) Create entry for `<kernel_parameter_list>`: `_sym_t sym`

(3) Create entry for `<data_structure_pointers>`, a pointer: `dtype (const) * _sym`

(4) Create entry for `<kernel_args_creation>` using above pointer and struct:
`_sym_t _sym_c = { _sym + _i * ncomp };`

(5) Create entry for `<kernel_call>`, add newly created struct instance to call arguments: `_sym_c`

end

As this example is not applying any shared memory techniques, we do not need to generate code to handle race conditions between threads as there is only one thread. Hence `ScalarArray` and `GlobalArray` data access can be achieved by passing pointers into the kernel, in Algorithm 16 we describe the creation of function arguments and parameters that allow the kernel to access these global data structures. Although we do not describe the code generation process for a shared memory execution model, our implementation is capable of producing thread-safe OpenMP code. In Listings 3.3 and 3.4 we present an example where a `ParticleDat` is assigned values from a `ScalarArray`.

Algorithm 16: Particle Loop code generation for `ScalarArrays` and `GlobalArrays`

Data: `ScalarArray` and `GlobalArray` instances $D^{(s)}$ and access descriptors $A^{(s)}$
Result: `<generated_structs>`, `<kernel_args_declaration>`, `<data_structure_pointers>`, `<kernel_args_creation>` and `<kernel_call>`
for $m \in \{0, \ldots, m^{(s)}\}$ **do**
  Construct identifier `sym` to use for temporary variables (usually the symbol used in the kernel)
  Identify underlying data type `dtype` from $d_m^{(s)}$
  Determine if the `const` qualifier is valid from $a_m^{(s)}$
  (1) Create entry for `<kernel_parameter_list>`, a pointer: `dtype (const) * sym`
  (2) Create entry for `<data_structure_pointers>`: `dtype (const) * sym`
  (3) Create entry for `<kernel_call>`, add pointer to call arguments: `sym`
end

Listing 3.3: *Example particle loop creation. We create a global 2-vector `S` and each particle `i` is assigned a 2-vector property `P`. The `ParticleLoop` copies the vector `S` into `P` for each particle.*

```
# setup removed for brevity
A.P = ParticleDat(ncomp=2)
S = ScalarArray(ncomp=2)

particle_loop = ParticleLoop(
    kernel=Kernel(
        name='plexample',
        code = """
        P.i[0] = S[0];
        P.i[1] = S[1];
        """
    ),
    dat_dict={
        'P': A.P(INC),
        'S': S(READ)
    }
)
```

Listing 3.4: *Example generated particle loop from input 3.3. This source code is compiled into a shared library such that the* `plexample_wrapper` *function can be called from Python.*

```
/* #### Structs generated per ParticleDat #### */
typedef struct
{
  double *restrict i;
}  _P_t;

/* #### Kernel function #### */
inline void k_plexample(double const *restrict S, _P_t P)
{
  P.i[0]=S[0];
  P.i[1]=S[1];
}

/* #### Library function ####
This is the entry point into the generated code from Python, the
number of particles and pointers to the data structures are passed
here.
*/
void plexample_wrapper(int const _N_LOCAL, double const *restrict S,
    double *restrict P)
{
  /* This is the main loop over particles. */
  for (int _i=0; _i<_N_LOCAL; _i++)
  {
    /* #### Kernel call arguments #### */
    _P_t P_c = { P+_i*2};
    /* #### Kernel call ####
    Here the user written kernel is called.
    */
    k_plexample(S,P_c);
  }
}
```

GPU

In Appendix A.8.1 we describe the process we use to generate GPU *Particle Loop* code. Our approach assigns one GPU thread to each particle in the order in which particle data is arranged in memory, this assumes that there are enough particles to fully occupy the GPU. With this approach, we guarantee that there is no memory access contention between particle data and by assigning GPU threads in the same order as particle data we obtain the most efficient data access pattern for particle data.

The GPU is a shared memory environment, hence we need to generate code that correctly implements the increment operation for `GlobalArray` instances. We use CUDA intrinsic functions to communicate data between GPU threads and to atomically increment

values in device memory, an example of this process is presented in Listing A.2 in the appendix.

3.2.2 Local Particle Pair Loop

We provide an overview of the code generation process for neighbour list based implementations of *Local Particle Pair Loop*. The neighbour lists we described are all utilised by first looping over all particles i then, for each particle i, looping over neighbours j. We mandated in our abstraction that data from particle j can only be read by the kernel K, hence our code generation system need only provide read access to this data.

We now describe the additions to the *Particle Loop* code generation system to implement the *Local Particle Pair Loop* assuming that on each architecture a suitable neighbour list has been constructed. Access to global data stored in `ScalarArray` and `GlobalArray` objects is identical to the *Particle Loop* case. As in the *Particle Loop* case we describe the code generation process for CPU architectures, CUDA code generation for GPU architectures is described in Appendix A.8.2.

We assume that the neighbour list exists as a sequential neighbour list as described in section 3.1.5. For N particles this neighbour list exists as an array $\vec{S} \in \mathbb{N}^{N+1}$ of starting points (and one end point) and an array of neighbours $\vec{b}$. The CPU particle loop template in listing 3.2 is amended to form the pair loop template in Listing 3.5.

Listing 3.5: *Template for CPU particle pair loop using a sequential neighbour list.*

```
// Structs generated per ParticleDat
<generated_structs>

// kernel source wrapped in a function
// ParticleDats are passed using above generated structs
// ScalarArray and GlobalArrays are passed as pointers
inline void k_<kernel_name>(<kernel_parameter_list>){
    <kernel_source>
}

// Externally available function to be called from Python
void <kernel_name>_wrapper(
    const int _N_LOCAL,
    long const * _START_POINTS,
    int const * _NLIST,
    <data_structure_pointers>
){

    // loop over all owned particles
    for(int _i=0 ; _i<_N_LOCAL ; _i++){
        // loop over neighbours of particle _i
        for (long _k=_START_POINTS[_i]; _k<_START_POINTS[_i+1]; _k++){
            const int _j = _NLIST[_k];

            // create instances of generated structs
            // for each ParticleDat
            <kernel_args_creation>

            // call kernel
            k_<kernel_name>(<kernel_call>);
        }
    }
}
```

Algorithm 15 generates C structures to hold pointers to the data for particle i, we modify the structure to include a pointer to the data for particle j, these modifications are described in Listing 3.6. We must also modify the stage that creates the kernel function argument, this step must initialise the C structure with a pointer to the data of particle j in addition to the data of particle i.

Listing 3.6: *Difference between particle loop C structure and particle pair loop C structure for a* `ParticleDat` *of data type* `dtype` *and symbol* `sym`*.*

```
// Particle Loop variant
typedef struct
{
    dtype *i;
}  _sym_t;

// Pair Loop variant
typedef struct
{
    dtype *i;
    dtype *j;
}  _sym_t;
```

To demonstrate the output, Listing 3.7 contains the Python source code to define a *Local Particle Pair Loop* that for each particle counts the number of neighbouring particles within a distance of 2. The generated C code is presented in Listing 3.8.

Listing 3.7: *Example particle pair loop creation that counts neighbouring particles within a cutoff radius of 2.*

```
# setup removed for brevity
A.P = PositionDat(ncomp=3)
A.NC = ParticleDat(ncomp=1, dtype=c_int)

loop = PairLoop(
    kernel=Kernel(
        name='n_count',
        code = """
            double r0 = P.i[0] - P.j[0];
            double r1 = P.i[1] - P.j[1];
            double r2 = P.i[2] - P.j[2];
            if ( r0*r0 + r1*r1 + r2*r2) < 4.0){
                NC.i[0] += 1;
            }
        """
    ),
    dat_dict={
        'P': A.P(READ),
        'NC': A.NC(INC)
    },
    shell_cutoff=2.0
)
```

Listing 3.8: *Generated C code to count the neighbours of each particle within a radius 2.*

```
// Structs generated per ParticleDat
typedef struct
{
  int *i;
  int *j;
}  _NC_t;
typedef struct
{
  double const *i;
  double const *j;
}  _P_t;

// Kernel function
inline void k_n_count(_NC_t NC, _P_t P){
    double r0 = P.i[0] - P.j[0];
    double r1 = P.i[1] - P.j[1];
    double r2 = P.i[2] - P.j[2];
    if ((r0*r0 + r1*r1 + r2*r2) < 4.0){
        NC.i[0] += 1;
    }
}

// Library function
void n_count_wrapper(int const _N_LOCAL, long const *_START_POINTS, int
    const *_NLIST, int *NC, double const *P){
  for (int _i=0; _i<_N_LOCAL; _i++)
  {
    for (long _k=_START_POINTS[_i]; _k<_START_POINTS[_i+1]; _k++)
    {
      const int _j = _NLIST[_k];

      // Struct initialisation
      _NC_t NC_c = { NC+_i*1, NC+_j*1};
      _P_t P_c = { P+_i*3, P+_j*3};

      // Kernel call
      k_test_host_pair_loop_1(NC_c,P_c);
    }
  }
}
```

GPU

As in the *Particle Loop* case, we assign one GPU thread to each particle, which loops over all the neighbours of the particle. We do not exploit Newton's Third Law, hence there is no memory access contention between threads for particle data. To increment global data

we use inter-thread communication and atomic operations in a near identical process to the `ParticleLoop` case. This code generation process is described in Appendix A.8.2.

Further Code Generation Discussion

We have provided a description of how our code generation framework produces C code which is valid but potentially not performant. There are two main classes of optimisation techniques which we apply to improve performance of the generated code for *local particle pair loops*. The first class is to arrange particle data such that it may be accessed more efficiently by the compute device. On GPU architectures particle data can be grouped by cell such that the data of particles within a cell form a contiguous block in memory.

On modern CPU architectures the efficiency of the floating point unit is often dependent on the layout of the underlying data. If data is arranged in the incorrect layout for the vector instructions the compiler must emit instructions to permute the arrangement of input and output data, and this reordering must occur before "useful" floating point operations are performed. In a cell by cell approach we know a priori that all particle data from a pair of cells will be accessed repeatedly, hence we generate code that explicitly reorders the data from both cells into temporary arrays. The kernel is then launched over all pairs of particles from the two cells using the data in the correct layout. Finally, we generate code to move written data back into the global data structure. This approach requires a kernel that contains enough computational work to amortise the cost of data movement.

The second main class of optimisations we implement are architecture and to some extent compiler specific optimisations to generate efficient low-level instructions. For example, modern GPU hardware by NVIDIA contains a read-only texture cache which is shared between all GPU cores. Through inspection of the passed access descriptors we determine which data is read-only and generate CUDA code that indicates to the NVIDIA CUDA compiler that this data is potentially a suitable candidate for this cache.

On CPU hardware we exploit the auto-vectorisation capabilities of the compiler. The compiler reads in the generated C source code and performs internal analysis to determine which optimisations it views as legal. We are most interested in the analysis that determines if a loop may be performed with vector instructions, for this to occur the compiler must find no reason not use vector instructions in the given block of code. To access `GlobalArray` data in an incremental manner we often produce C code with a similar pattern to Listing 3.9 where a variable is incremented in each iteration of a loop. Some modern C compilers will refuse to generate vector instructions due to a detected loop dependence (even with restrict qualifiers), clearly there is a loop dependence, however, if we assume addition of floating point numbers is associative the loop can be performed in a vector manner. If we emit the C code in listing 3.10 we heuristically find compilers are more likely to vectorise the loop.

Listing 3.9: *Candidate loop where a compiler could refuse to emit vector instructions due to a loop dependance.*

```
void kernel(..., double * restrict a, ... ){
    for(int _j=0; _j<_jmax ; _j++){
        a[0] += ... ;
    }
}
```

Listing 3.10: *Candidate loop where a compiler could be persuaded to emit vector instructions.*

```
void kernel(..., double * restrict a, ... ){
    double at[0] = {0};
    for(int _j=0; _j<_jmax ; _j++){
        at[0] += ... ;
    }
    a[0] += at[0];
}
```

3.3 Results

This section is adapted from our published article *A domain specific language for performance portable molecular dynamics algorithms* [73]. To demonstrate the performance, portability and scalability of our code generation framework on two different mode HPC architectures, we implemented the Velocity Verlet integrator as described in Algorithms 1 and 17. We simulated a Lennard-Jones liquid system of non-bonded particles interacting via the potential in Equation (2.1) and parameters in Table 3.2. The C kernel for this interaction is presented in Listing 2.5 and is executed in a *Local Particle Pair Loop*. The position and velocity update stages of the Velocity Verlet Algorithm are performed via *Particle Loops* with kernels given in Listings 3.11 and 3.12. A summary of all looping operations and access descriptors is given in Table 3.1. The full source code can be found in the examples[1] subdirectory of [71]. The same code can be used to run the simulation both on a CPU and a GPU if the appropriate definitions shown in listing 2.2 are added at the beginning of the Python code.

[1] `code/examples/lennard-jones`

Algorithm 17: Velocity Verlet integrator used in Section 3.3. The system is integrated numerically with a time step of size δt until the final time $T = n_{\max}\delta t$.

Create `ParticleDat`s for forces $\vec{F}$ and velocities $\vec{v}$.
Create `PositionDat` for particle positions.
Initialise particle positions and velocities.
Collect `ParticleDat`s and `PositionDat` in a `State` object
for *timestep* $i = 1, \ldots, n_{\max}$ **do**
  For all particles i: $\vec{v}^{(i)} \mapsto \vec{v}^{(i)} + \frac{\delta t}{2m}\vec{F}^{(i)}$, $\vec{r}^{(i)} \mapsto \vec{r}^{(i)} + \delta t\vec{v}^{(i)}$
  For all pairs (i, j): $\vec{F}^{(i)} \mapsto \vec{F}^{(i)} + \vec{f}(\vec{r}^{(i)}, \vec{r}^{(j)})$
  For all particles i: $\vec{v}^{(i)} \mapsto \vec{v}^{(i)} + \frac{\delta t}{2m}\vec{F}^{(i)}$
end

Operation	Loop type & kernel	Access Descriptors
$\vec{v}^{(i)} \mapsto \vec{v}^{(i)} + \frac{\delta t}{2m}\vec{F}^{(i)}$ $\vec{r}^{(i)} \mapsto \vec{r}^{(i)} + \delta t\vec{v}^{(i)}$	`ParticleLoop` Listing 3.11	$\vec{v}$ [`INC`], $\vec{r}$ [`INC`], $\vec{F}$ [`READ`]
$\vec{F}^{(i)} \mapsto \vec{F}^{(i)} + \vec{f}(\vec{r}^{(i)}, \vec{r}^{(j)})$	`PairLoop` Listing 2.5	$\vec{F}$ [`INC_ZERO`], $\vec{r}$ [`READ`]
$\vec{v}^{(i)} \mapsto \vec{v}^{(i)} + \frac{\delta t}{2m}\vec{F}^{(i)}$	`ParticleLoop` Listing 3.12	$\vec{v}$ [`INC`], $\vec{F}$ [`READ`]

Table 3.1: *Access descriptors for the loops in the Velocity Verlet Algorithm 17.*

Listing 3.11: *Velocity and position update kernel in the Velocity Verlet Algorithm 17. The constants* `dt` *and* `dht_iMass` *are set to* δt *and* $\delta t/(2m)$ *and passed to the pairloop as* `Constant` *objects.*

```
v.i[0] += F.i[0]*dht_iMASS;
v.i[1] += F.i[1]*dht_iMASS;
v.i[2] += F.i[2]*dht_iMASS;
r.i[0] += dt*v.i[0];
r.i[1] += dt*v.i[1];
r.i[2] += dt*v.i[2];
```

Listing 3.12: *Velocity update kernel in the Velocity Verlet Algorithm 17. As in Listing 3.11, the quantity* $\delta t/(2m)$ *is passed to the pairloop as a* `Constant` *object to replace* `dht_iMASS`*.*

```
v.i[0] += F.i[0]*dht_iMASS;
v.i[1] += F.i[1]*dht_iMASS;
v.i[2] += F.i[2]*dht_iMASS;
```

3.3.1 Comparison To Other Codes

To verify that the code generation approach does not introduce any sizable computational overheads, we compare the performance of our code to monolithic C/Fortran implementations in well established and optimised MD libraries. For this we performed the same strong scaling experiment with DL_POLY (version 4.08), LAMMPS (release dated 1st March 2016) and our code generation framework (subdirectory `release` of [71]). A strong scaling experiment investigates how the time to solution for a given computational task is reduced by increasing the computational resource used. Here the computational task is a simulation containing a significant number of particles that interact with non-bonded interactions. Raw results can be found in the accompanying data repository [72].

All codes were built with the Intel 2016 compiler suite and OpenMPI 1.8.4 (with the exception of DL_POLY, which used OpenMPI 2.0.0). The NVIDIA CUDA toolkit version 7.5.18 was used for the GPU compilation and the framework was run with Python 2.7.8. The numerical experiments were carried out on the University of Bath HPC facility "Balena". All nodes of the cluster consist of two Intel Xeon E5-2650v2 (2.6GHz) processors with eight cores each; in addition some nodes are equipped with Nvidia Tesla K20X GPU accelerator cards. As the GPU port of LAMMPS offloads the force calculation, we allowed LAMMPS to use all 16 cores of the host CPU along with the GPU. In contrast, in our framework the entire simulation is run on the GPU and it is sufficient to use a single MPI rank which acts as the host controller.

Parameter	Value
Number of atoms: N	10^6
Number of time steps: $n_{\max}$	10^4
Number density: ρ	0.8442
Force cutoff: r_c	2.5
Force extended cutoff: $r_n = r_c + \delta$	2.75
Steps between neighbour list update:	$20^\dagger$

Table 3.2: *Parameters of Lennard-Jones benchmark for the strong scaling experiment; units are chosen such that $\sigma = \epsilon = 1$ († = excluding DL_POLY, see main text).*

We use the parameters in Table 3.2, adapted from a LAMMPS benchmark [65]. All three codes implement the neighbour list method for force calculations. For LAMMPS and our framework the extended cutoff $r_n = r_c + \delta$ was chosen such that $\delta = 0.1r_c$ with a neighbour list update every 20 iterations. In contrast, DL_POLY automatically updates the neighbour-list when necessary.

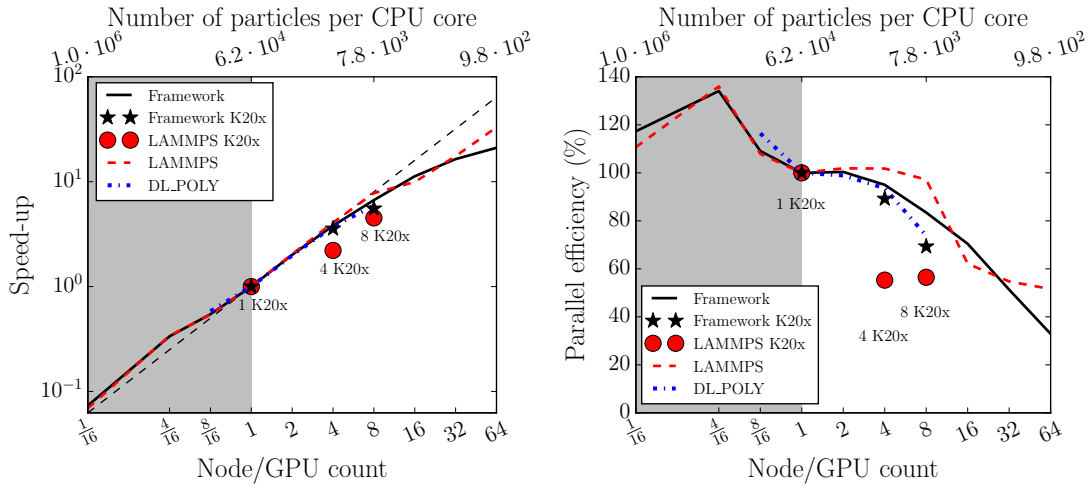

Figure 3-9: *Strong scaling experiment: parallel speed-up (left) and parallel efficiency (right) for the time taken (s) to compute $n_{max} = 10^4$ Velocity Verlet iterations of $N = 10^6$ particles using DL_POLY, LAMMPS and our implementation (labeled as "Framework"). Efficiency and speed-up are relative to one full node (16 cores). Efficiency is calculated according to Equation (3.9). In the left plot perfect scaling is indicated by the dashed gray line. Raw results are presented in Table 3.3 and simulation parameters are tabulated in Table 3.2.*

Node/GPU count	Integration Time (Seconds)				
	Framework		LAMMPS		DL_POLY_4
	CPU	GPU	CPU	GPU	CPU
1/16	$6.83 \cdot 10^3$		$8.22 \cdot 10^3$		
4/16	$1.49 \cdot 10^3$		$1.67 \cdot 10^3$		
8/16	$9.18 \cdot 10^2$		$1.05 \cdot 10^3$		$4.99 \cdot 10^3$
1	$5.01 \cdot 10^2$	$3.85 \cdot 10^2$	$5.69 \cdot 10^2$	$2.75 \cdot 10^2$	$2.91 \cdot 10^3$
2	$2.50 \cdot 10^2$		$2.79 \cdot 10^2$		$1.47 \cdot 10^3$
4	$1.32 \cdot 10^2$	$1.08 \cdot 10^2$	$1.40 \cdot 10^2$	$1.24 \cdot 10^2$	$7.76 \cdot 10^2$
8	$7.50 \cdot 10^1$	$6.95 \cdot 10^1$	$7.32 \cdot 10^1$	$6.08 \cdot 10^1$	$4.92 \cdot 10^2$
16	$4.45 \cdot 10^1$		$5.72 \cdot 10^1$		
32	$3.05 \cdot 10^1$		$3.25 \cdot 10^1$		
64	$2.38 \cdot 10^1$		$1.72 \cdot 10^1$		

Table 3.3: *Strong scaling experiment: time taken (s) to compute $n_{max} = 10^4$ Velocity Verlet iterations of $N = 10^6$ particles using DL_POLY, LAMMPS and our implementation (labeled as "Framework"). Further simulation parameters are given in Table 3.2. CPU nodes consist of two eight core E5-2650v2 CPUs, GPU nodes contain one or more K20X GPUs. GPUs are compared against CPU nodes on a one-to-one basis. For GPU results, the Framework used one CPU core as a host for each GPU. LAMMPS implements a GPU offload approach and hence used all CPU cores on the node in addition to available GPUs. All codes were built with the Intel 2016 compiler suite and OpenMPI 1.8.4 (with the exception of DL_POLY, which used OpenMPI 2.0.0). The NVIDIA CUDA toolkit version 7.5.18 was used for the GPU compilation and the framework was run with Python 2.7.8.*

The total integration time on up to 1024 cores (64 nodes) and up to 8 GPUs is tabulated

in Table 3.3. Parallel speed-up and parallel efficiency are plotted in Figure 3-9; grey regions indicate core counts contained within a single CPU node. On the largest core count (1024 cores) the average local problem size is reduced to 1,000 particles per processor. To provide a fair comparison, one K20X GPU is compared to a full 16-core CPU node since in this case the power consumption is comparable (235 W for the K20X GPU [56] vs. $2 \times 95\,\mathrm{W}$ +(memory power consumption) for the Intel Xeon E5-2650v2 CPU [40]). We write $t(p, N)$ for the measured wall clock time required to integrate a system with N particles on p CPU nodes or GPUs. The corresponding speed-up and parallel efficiency (relative to one CPU node or one GPU) are defined as

$$\begin{aligned} \text{Speed-up} &= \frac{t(1,N)}{t(p,N)} \\ \text{Strong parallel efficiency} &= \frac{t(1,N)}{p \times t(p,N)} \end{aligned} \tag{3.9}$$

and shown in Figure 3-9.

The absolute times demonstrate that the framework provides comparable performance and scalability to DL_POLY and LAMMPS. In fact we find that for this particular setup both LAMMPS and our code are significantly faster than DL_POLY and scale better. It should be kept in mind, however, that currently both LAMMPS and DL_POLY have a much wider range of applications and provide functionality which is not yet implemented in our framework. A socket-to-socket comparison demonstrates that one full GPU can only deliver a slightly higher performance than a full CPU node. Again, the same is observed for LAMMPS.

To test performance for very large problem sizes we also carried out a weak scaling experiment. In a weak scaling experiment the average work per unit computational resource is fixed and the total problem size grows proportional to the number of nodes. A system with $512,000$ particles per CPU core ($8,192,000$ particles per node) was integrated over 5000 timesteps. For the largest computational configuration (1024 cores) the total problem size is about half a billion ($5.24 \cdot 10^8$) particles. All other system parameters are unchanged from Table 3.2. The total time for increasing problem sizes is shown in Figure 3-10 (left). The weak parallel efficiency is defined as

$$\text{Weak parallel efficiency} = \frac{t(1,N)}{t(p, N \cdot p)} \tag{3.10}$$

and plotted in Figure 3-10 (right). We observe that (relative to one node) the parallel efficiency never drops below 90% and conclude that the framework will effectively scale to systems containing very large numbers of particles on a significant core count.

The number of particles on a single CPU node in the previous weak scaling run is too large to fit into GPU memory. To also compare the weak scalability of the generated CPU and GPU code we therefore repeat the same experiment with a reduced number of 512,000 particles per node. The resulting time and parallel efficiency are shown in

Node count	Integration Time (10^3 Seconds)
1/16	1.61
2/16	1.65
4/16	1.66
8/16	1.52
1	1.91
2	1.93
4	1.94
8	1.96
16	1.99
32	2.01
64	2.09

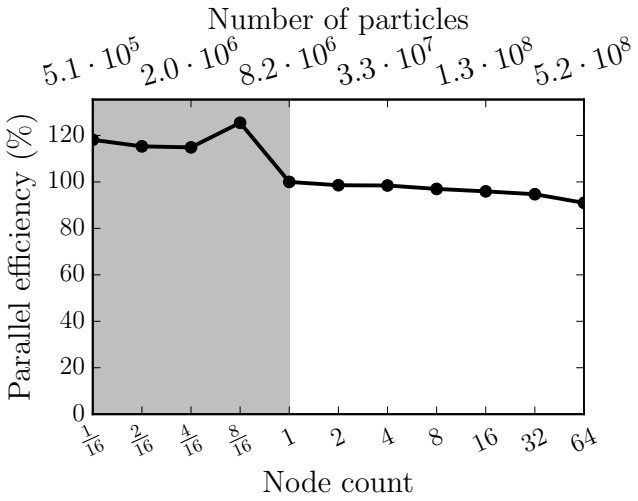

Figure 3-10: *CPU-only weak scaling experiment: time taken to integrate the system over* $n_{\max} = 5000$ *time steps (left) and parallel efficiency (right). The efficiency relative to one full node (right) is calculated according to Equation (3.10). The top horizontal axes shows the total number* N *of particles in the system; the number of particles per core is kept fixed at* $512,000$ *(*$8,192,000$ *particles per node).*

Figure 3-11. While the parallel efficiency is worse for the GPU, it never drops below 60%. On one node the GPU code is about twice as fast as the CPU code and on 16 nodes this speedup factor drops to around 1.3×. This can be explained by the fact that on one node the CPU implementation is slower and therefore communication overheads will have a relatively larger impact on the GPU code. To improve scalability further, we will investigate overlapping communication and communication in the future. This, however, is usually more challenging on GPUs due to the reduced work in halo regions.

Node/GPU count	Integration Time (Seconds)	
	CPU	GPU
1	116.9	60.3
4	123.2	78.0
8	124.8	89.7
16	129.9	94.1

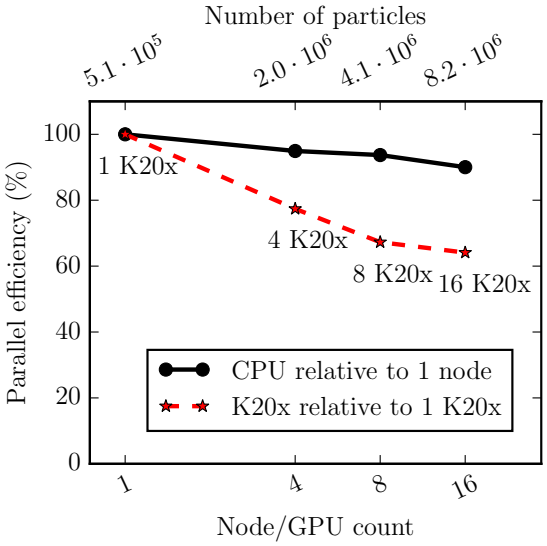

Figure 3-11: *CPU-GPU weak scaling experiment with reduced particle number: time taken to simulate* $n_{\max} = 5000$ *time steps (left) parallel efficiency relative to a single GPU/node, calculated according to Equation (3.10) (right). The number of particles per node is kept fixed at 512,000.*

	Intel Xeon node		K20X GPU	
kernel	peak	time	peak	time
Force	16.5%	54.8%	11.9%	36.9%
Force & PE	7.5%	6.5%	14.3%	2.6%

Table 3.4: *Absolute performance metrics (as percentage of peak performance and integration time) for two kernels recorded from GPU weak scaling experiment presented in Figure 3-11. The "Force & PE" kernel is only called every 10 iterations and hence accounts for a smaller proportion of the total runtime than the "Force" kernel.*

Absolute performance

To quantify the absolute performance on both CPU and GPU we use data collected in the second weak scaling experiment (see Figure 3-11). The computationally most expensive operation in the simulation is the force update step performed with a particle pair loop. This accounts for 54.8% of the total runtime on the CPU and 36.9% on the GPU. As in this simulation the potential energy was updated every 10 iterations, we also report performance metrics for the combined force- and potential-energy (PE) update.

With the vector instruction set each core of an E5-2650v2 (2.6 GHz) Intel CPU can perform 4 double precision additions and 4 double precision multiplications per clock cycle, resulting in a total performance of 332.8 GFLOPs per node. The peak double precision floating point performance of the nVidia Tesla K20x GPU is quoted as 1.31 TFLOPs [58].

Absolute performance numbers for a single-node run are reported in Table 3.4. The measured times only include the time spent in the auto-generated C code, but we found that the launch of a shared library function from Python has a negligible overhead ($\approx$ 10–20μs). Since the system is spatially homogeneous and there is little load imbalance, we report measurements collected by a single core on the fully populated node. The results demonstrate that the computationally most relevant kernels use a significant fraction of the peak floating point performance. As confirmed by the report generated by the compiler, the kernel for the Lennard-Jones force calculation in Listing 2.5 is automatically vectorised.

3.3.2 Structure Analysis Algorithms

We present the performance of the structure analysis algorithms described in Sections 1.1.4 and 2.3.2 implemented within our framework.

For this we first add an on-the-fly implementation of the BOA analysis method. This is achieved by extending the main timestepping loop in Algorithm 17 by calls to the `PairLoop` and `ParticleLoop` which evaluate Q_ℓ according to Algorithms 2 and 3. The source code is available in the `examples/on-the-fly-analysis` subdirectory of [71].

To initialise the simulation, 125000 identical particles are arranged in a periodic cubic lattice and their velocities are sampled from a normal distribution. After allowing the system to equilibrate for 50,000 steps in an microcanonical ensemble we coupled the system

to an Andersen thermostat with a target temperature near zero for 500,000 iterations. The final configuration consists of two distinct regions. The first is void of particles while the second contains a crystal structure. Figure 3-12 shows the change of Q_4, Q_5 and Q_6 throughout the simulation.

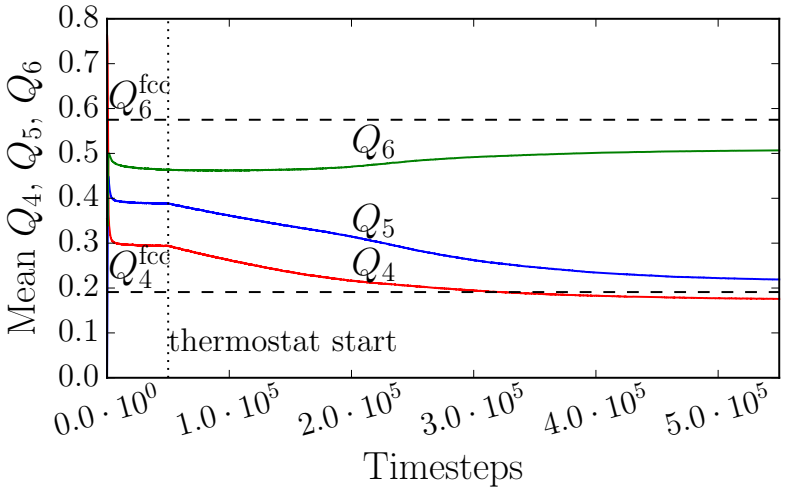

Figure 3-12: *Evolution of mean Q_4, Q_5 and Q_6 values over the course of the simulation. The horizontal dashed lines plot the expected Q_4 and Q_6 values of a perfect FCC lattice.*

A distribution of the Q_4 and Q_6 values at the final time is shown in Figures 3-13 and 3-13. This distribution describes the proportion of FCC and HCP in the final configuration as classified by the BOA method. We purely focus on the implementation of the method and do not attempt a physical interpretation of the results.

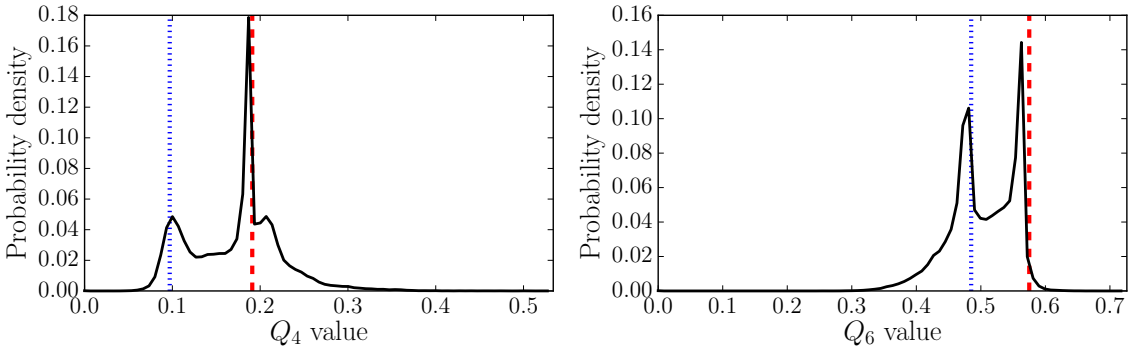

Figure 3-13: *Probability density of Q_4 values (left) and Q_6 values (right) in final system configuration. (left) Dashed vertical line at $Q_4 = 0.097$ is the expected Q_4 value of a perfect hcp lattice. Dashed vertical line at $Q_4 = 0.191$ is the expected Q_4 value of a perfect fcc lattice. (right) Dashed vertical line at $Q_6 = 0.485$ is the expected Q_6 value of a perfect hcp lattice. Dashed vertical line at $Q_6 = 0.575$ is the expected Q_6 value of a perfect fcc lattice.*

To demonstrate that the resulting code still scales well in parallel, we carry out a weak scaling experiment with the parameters in Table 3.5. The results are shown in Figure 3-14 and confirm that adding the on-the-fly analysis and thermostat have no negative impact

Parameter	Value
Number of atoms per node:	524288
Number of time steps: $n_{\max}$	5000
Non-dimensionalised density: ρ	0.8442
Force cutoff: r_c	3.0
Force extended cutoff: $\overline{r}_c = r_c + \delta$	3.3
Steps between neighbour list updates:	18

Table 3.5: *Parameters of bond order analysis weak scaling experiment. Units are chose such that* $\sigma = \epsilon = 1$.

Node count	Integration Time (10^2 Seconds)
1/16	4.37
2/16	4.48
4/16	4.50
8/16	4.60
1	4.99
2	5.03
4	5.09

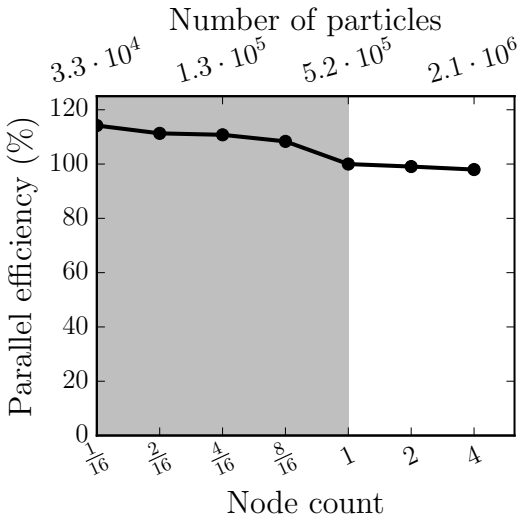

Figure 3-14: *Weak scaling experiment that combines a simulation with on-the-fly analysis. Time taken to integrate 5000 steps, parallel efficiency relative to a single node (right).*

on parallel efficiency.

Finally, the common neighbour analysis was implemented as a parallel post-processing step. C-Kernels for Algorithms 4, 5, 6 and 25 can be found in the examples[2] subdirectory of [71]. We validated our implementations by verifying that perfect crystals are correctly classified in each of the FCC, BCC and HPC configurations. We then applied the method to the test case with 125000 particles mentioned above. For the final configuration the algorithm classified 19360 (15.5%) particles as FCC and 13052 (10.4%) particles as HCP while 92588 (74.1%) particles were left unclassified.

[2] `examples/structure/cna`

CHAPTER 4

MODERN ALGORITHMS FOR ELECTROSTATIC INTERACTIONS

4.1 Introduction

Alongside the short-range interactions that are used in applications such as atomistic or molecular modelling, it is typically necessary to also consider the charges that particles carry. The electrostatic interactions between charged particles cannot generally be computed efficiently via the pairwise operations we describe in earlier chapters. These electrostatic interactions form an important component of the underlying physical properties of many materials and cannot be neglected.

A detailed description of the theory of electrostatics is given by Jackson [43]. Here we provide a brief overview of the electrostatic theory required for MD and discuss two existing methods. We present our parallel implementations of these two methods in Chapter 5. For simplicity the equations we state and derive are written in Gaussian units, the conversion between Gaussian units and Systéme international (SI) units is straightforward and an overview is given in Appendix A.3. We begin by defining the electric field $\vec{E}$ as the force exerted per unit charge at a point in space. Hence if a particle resides at a position $\vec{r}$ and carries a net charge q the force $\vec{F}$ exerted on the particle by the electric field $\vec{E}$ is given by

$$\vec{F} = q\vec{E}(\vec{r}). \tag{4.1}$$

Consider a system consisting only of a pair of particles (i, j) at positions $(\vec{r}_i, \vec{r}_j)$ that carry charges (q_i, q_j). We can write the force exerted on the first particle $\vec{F}_i$ in terms of the electric field induced by the second particle $\vec{E}_j$,

$$\vec{F}_i = q_i\vec{E}_j(\vec{r}_i). \tag{4.2}$$

By considering each charged particle as a point-wise object the form of $\vec{E}_j$ is given by

Coulomb's Law,

$$\vec{E}_j(\vec{r}) = \frac{q_j}{|\vec{r} - \vec{r}_j|^2} \frac{\vec{r} - \vec{r}_j}{|\vec{r} - \vec{r}_j|}. \tag{4.3}$$

For a general charge density $\rho(\vec{r})$ the induced electric field $\vec{E}$ can be described by the differential form of Gauss' Law of electrostatics,

$$\vec{\nabla} \cdot \vec{E}(\vec{r}) = 4\pi\rho(\vec{r}). \tag{4.4}$$

Furthermore, the electric field can be written as the gradient of a potential field known as the electric field potential ϕ,

$$\vec{E}(\vec{r}) = -\vec{\nabla}\phi(\vec{r}). \tag{4.5}$$

By combining the differential form of Gauss' Law (4.4) with equation (4.5) we deduce that the electric field potential $\phi(\vec{r})$ induced by a charge density $\rho(\vec{r})$ is the solution of Poisson's equation:

$$-\vec{\nabla} \cdot (\vec{\nabla}\phi(\vec{r})) = -\Delta\phi(\vec{r}) = 4\pi\rho(\vec{r}), \tag{4.6}$$

with free space boundary conditions, i.e. the system is surrounded by an infinite vacuum such that $\phi(\vec{r}) \to 0$ as $|\vec{r}| \to \infty$.

We can recover Coulomb's Law from equation (4.6) by considering a charge density ρ that describes a point-wise particle at the origin with net charge q:

$$-\Delta\phi(\vec{r}) = 4\pi q\delta(\vec{r}), \tag{4.7}$$

where δ is the Dirac delta function. The solution ϕ to equation (4.7) is proportional to the fundamental solution of the Laplace equation in three dimensions and is given by

$$\phi(\vec{r}) = \frac{q}{|\vec{r}|}. \tag{4.8}$$

We refer to ϕ as given by equation (4.8) as the Coulomb potential. Using the Coulomb potential the electrostatic potential energy between a pair of particles (i, j) that are separated by distance r_{ij} and carry charges (q_i, q_j) is given by

$$U_{ij} = \frac{q_i q_j}{r_{ij}}. \tag{4.9}$$

With free space boundary conditions, calculating the inter-particle interactions for all $N(N-1) = \mathcal{O}(N^2)$ particle pairs, by using Coulomb's Law, would be a sufficient though inefficient method to compute inter-particle forces and potential energies. Typically, simulation domains are combined with periodic boundary conditions where application of Coulomb's Law results in a conditionally convergent summation.

4.1.1 Coulomb Potential Truncation

When considering inter-particle interactions via short-range potentials we are able to ignore interactions between pairs of particles which are separated by sufficiently large distances. Ignoring pairs of sufficiently well separated particles is equivalent to truncating the potential to zero. In an infinite system the error incurred from the truncation of short-range potentials is bounded as the functional form of these potentials decays to zero sufficiently rapidly as inter-particle distance increases.

The Coulomb potential exhibits a functional form proportional to inverse distance $(1/r)$, as r increases the magnitude of the potential decays to zero. We demonstrate that as the inter-particle distance r increases the Coulomb potential does not decay to zero at high enough rate to make a truncation without incurring an unacceptable error.

To demonstrate the effect of a truncation of the Coulomb potential we consider a spherical system S centred at the origin. Within the sphere S of radius L we place a constant charge density $\rho_0 = 1$, this is an approximation of a uniformly distributed collection of charged particles. Outside the sphere we set the charge density to zero, this represents a vacuum absent of charges.

We investigate the error induced by truncating potentials of the form $|\vec{r}|^{-\beta}$ for $\beta \in \mathbb{N}$ at the centre of the sphere. We consider $\beta > 1$ as these values of β correspond to higher order moments, such as dipole moments in the $\beta = 2$ case and quadrupole moments in the $\beta = 3$ case, and these moments are relevant in our Fast Multipole Method discussions in later sections. The potential U_β at the centre of the sphere is given by

$$U_\beta(\vec{0}) = \int_S \frac{\rho_0}{|\vec{r}|^\beta} d\vec{r}, \tag{4.10}$$

and we recover the electrostatic potential U when $\beta = 1$. If we truncate the potential U_β at a radius $r_c = L - a$ then we ignore the contribution to the potential at the origin from the charge density contained in a shell of width a at the sphere boundary.

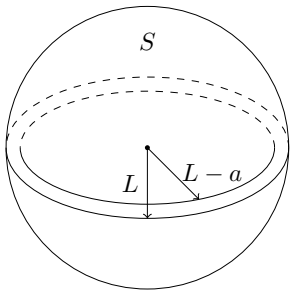

Figure 4-1: *Spherical system S of radius L and truncation radius $r_c = L - a$.*

The error in the potential ϵ_U at the centre of the sphere due to the truncation of the

Coulomb potential at a radius $r_c = L - a$ is given by an integral over the excluded shell:

$$\epsilon_U = \int_0^{2\pi} d\phi \int_0^{\pi} \sin(\theta) d\theta \int_{r_c}^{L} \frac{1}{r^\beta} r^2 dr \tag{4.11}$$

$$= \begin{cases} \frac{4\pi}{3-\beta} r_c^{3-\beta} \left(\left(\frac{L}{r_c} \right)^{3-\beta} - 1 \right) & \text{if } \beta \neq 3 \\ 4\pi \log \left(\frac{L}{r_c} \right) & \text{if } \beta = 3 \end{cases} \tag{4.12}$$

We are interested in the behaviour when $L/r_c \to \infty$ as this corresponds to an increasing system size with an interaction cutoff r_c that is much smaller than the system extent. Equation (4.12) demonstrates that for $\beta < 4$ the error ϵ_U is unbounded and grows in this limit. Furthermore, when $\beta \geq 4$ the error ϵ_U is suppressed by the $r_c^{3-\beta}$ term. We conclude that the electrostatic interactions in infinite systems cannot be computed by a truncated Coulomb potential and now discuss alternative approaches.

4.2 Particle Ewald Summation

This section follows the Particle Ewald Summation discussion by Frenkel and Smit in [26]. Particle Ewald [23, 26] summation is a technique to compute the long-range electrostatic potential energy and forces arising from Coulombic interactions between charged particles. The technique is applicable to a set of charged particles contained within a simulation domain with periodic boundary conditions. Elements of this section are published in the conference proceedings [74] alongside a parallel implementation. We describe the technique for a cubic simulation cell of side length L, however, the method is readily generalised to cuboid simulation cells and parallelepiped shaped simulation domains. The method offers reasonable performance for small to medium sized systems ($N \approx 10^3$-10^4 particles) with a $\mathcal{O}(N^{3/2})$ computational complexity.

As discussed in Section 4.1, the Coulombic potential ϕ at a point in the domain is the solution of the equation

$$-\Delta\phi(\vec{r}) = 4\pi\rho(\vec{r}), \tag{4.13}$$

$$\rho(\vec{r}) = \sum_{\vec{\mathfrak{n}} \in \mathbb{Z}^3} \sum_{j=1}^{N} q_j \delta(\vec{r} - \vec{r}_j - L\vec{\mathfrak{n}}). \tag{4.14}$$

The charge density $\rho(\vec{r})$ is formed by considering the charge q_j of particle j to exist at a single point $\vec{r}_j$. The summation with index $\mathfrak{n}$ duplicates and translates the charge density of the primary simulation cube to each periodic image. The solution ϕ to Equations (4.13, 4.14) is a scalar field which is periodic in $\mathbb{R}^3$ with a period given by the extents of the simulation cell.

Conceptually the Ewald method splits the calculation into two main components by rewriting the charge density of each particle as the sum of two terms. The two terms, which sum to give the original charge density, can be treated separately and the results

recombined using the superposition principle. More formally, particle j at position $\vec{r}_j$ and total charge q_j has a charge density given by $q_j\delta(\vec{r}-\vec{r}_j)$. We split this charge density into two terms by adding and subtracting a Gaussian of width $\propto \alpha^{-1/2}$ and charge density $S_\alpha(\vec{r})$:

$$S_\alpha(\vec{r}) = \left(\frac{\alpha}{\pi}\right)^{3/2} \exp\left(-\alpha|\vec{r}|^2\right) \tag{4.15}$$

$$\delta(\vec{r}) = D^{(\mathrm{sr})}(\vec{r}) + D^{(\mathrm{lr})}(\vec{r}), \text{ where} \tag{4.16}$$

$$D^{(\mathrm{sr})}(\vec{r}) = \delta(\vec{r}) - S_\alpha(\vec{r}), \tag{4.17}$$

$$D^{(\mathrm{lr})}(\vec{r}) = S_\alpha(\vec{r}), \tag{4.18}$$

a one dimensional representation of the splitting process is given in Figure 4-2.

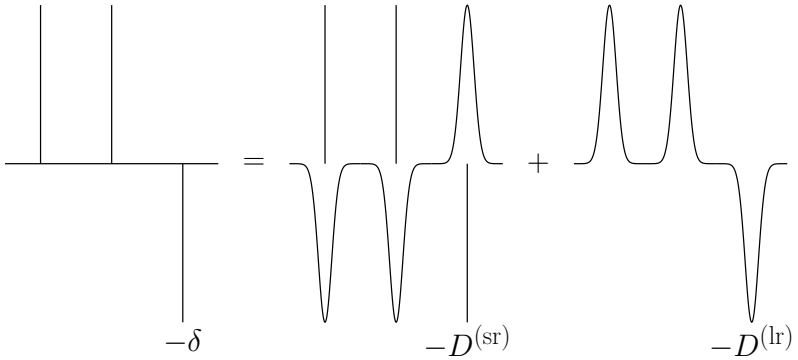

Figure 4-2: *One dimensional representation of the charge splitting process for two positive charges and one negative charge. The $-\delta$,$-D^{(sr)}$ and $-D^{(lr)}$ labels the figure indicate the charge splitting process for the right-hand charge.*

As Poisson's equation is linear, the total potential ϕ is given by the sum of a short-range potential $\phi^{(\mathrm{sr})}$ and a long-range potential $\phi^{(\mathrm{lr})}$ where

$$-\Delta\phi^{(\mathrm{sr})}(\vec{r}) = 4\pi \sum_{\vec{\mathfrak{n}}\in\mathbb{Z}^3} \sum_{j=1}^{N} q_j D^{(\mathrm{sr})}(\vec{r}-\vec{r}_j - L\mathfrak{n}), \tag{4.19}$$

$$-\Delta\phi^{(\mathrm{lr})}(\vec{r}) = 4\pi \sum_{\vec{\mathfrak{n}}\in\mathbb{Z}^3} \sum_{j=1}^{N} q_j D^{(\mathrm{lr})}(\vec{r}-\vec{r}_j - L\mathfrak{n}). \tag{4.20}$$

First we focus on the short-range potential $\phi^{(\mathrm{sr})}$ where we consider the potential field induced by a single unit charge at the origin. We separately compute the potential induced by a delta function charge density and the potential induced by a Gaussian charge density. We define the short-range potential $\phi^{(\mathrm{sr})}$ for a single unit charge as the sum of these two potentials.

As discussed in the introduction, we can use the fundamental solution of Poisson's equation to write down the contribution to $\phi^{(\mathrm{sr})}$ from the delta function term in $D^{(\mathrm{sr})}$. If a unit charge is positioned at the origin then the Coulomb potential induced by the delta

function is given by

$$-\Delta\phi(\vec{r}) = 4\pi\delta(\vec{r}), \tag{4.21}$$

with solution

$$\phi(\vec{r}) = \frac{1}{|\vec{r}|}. \tag{4.22}$$

The second contribution to $\phi^{(\mathrm{sr})}$ is the potential induced by a Gaussian shaped charge density S_α centred at the origin with unit volume. We consider only the radial component of the Poisson's equation by exploiting the rotational symmetry of the Gaussian charge density:

$$-\Delta\phi(\vec{r}) = 4\pi S_\alpha^{(\mathrm{sr})}(\vec{r}) \tag{4.23}$$

$$\implies -\frac{1}{r}\frac{\partial^2}{\partial r^2} r\phi(\vec{r}) = 4\pi \left(\frac{\alpha}{\pi}\right)^{3/2} \exp\left(-\alpha|\vec{r}|^2\right) \tag{4.24}$$

$$\implies -\frac{\partial}{\partial r} r\phi(\vec{r}) = 4\pi \int_\infty^r r' \left(\frac{\alpha}{\pi}\right)^{3/2} \exp\left(-\alpha r'^2\right) dr' \tag{4.25}$$

$$= -2\left(\frac{\alpha}{\pi}\right)^{\frac{1}{2}} \exp\left(-\alpha r^2\right) \tag{4.26}$$

$$\implies r\phi(\vec{r}) = 2\left(\frac{\alpha}{\pi}\right)^{\frac{1}{2}} \int_0^r \exp(-\alpha r'^2) dr'. \tag{4.27}$$

Using the standard error function defined as

$$\mathrm{erf}(x) = \frac{2}{\sqrt{\pi}} \int_0^x e^{-t^2}\, dt, \tag{4.28}$$

we can write the potential field induced by a Gaussian charge density as

$$\phi(\vec{r}) = \frac{\mathrm{erf}(\sqrt{\alpha}|\vec{r}|)}{|\vec{r}|}. \tag{4.29}$$

The construction of the short-range charge density $D^{(\mathrm{sr})}$ places a delta function and an opposing Gaussian function at the position $\vec{r}_j$ of each charge. We can construct the pairwise short-range potential for each charge as the sum of the potential induced by the delta function and the potential induced by the Gaussian function. If there exists charge q_j with position $\vec{r}_j$ then the short-range charge density is

$$\rho_j^{(\mathrm{sr})}(\vec{r}) = q_j(\delta(\vec{r} - \vec{r}_j) - S_\alpha(\vec{r} - \vec{r}_j)) \tag{4.30}$$

which induces a short-range potential field

$$\phi_j^{(\mathrm{sr})}(\vec{r}) = q_j \left(\frac{1}{|\vec{r} - \vec{r}_j|} - \frac{\mathrm{erf}(\sqrt{\alpha}|\vec{r} - \vec{r}_j|)}{|\vec{r} - \vec{r}_j|} \right) \tag{4.31}$$

$$= q_j \frac{\mathrm{erfc}(\sqrt{\alpha}|\vec{r} - \vec{r}_j|)}{|\vec{r} - \vec{r}_j|}, \tag{4.32}$$

$$\text{where} \quad \mathrm{erfc}(x) = \frac{2}{\sqrt{\pi}} \int_x^\infty e^{-t^2}\, dt = 1 - \mathrm{erf}(x). \tag{4.33}$$

A potential of the form of equation (4.32) decays to zero exponentially quickly with rate α which allows a truncation at some computationally reasonable inter-particle distance. A plot of the short-range potential for a unit charge can be found in Figure 4-3.

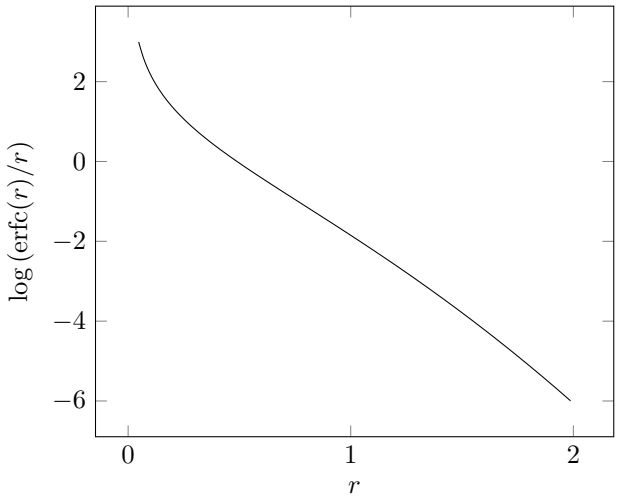

Figure 4-3: *Log-scale plot of the short-range potential induced by a unit charge at the origin.*

We now derive the long-range potential induced by a charge density constructed via $D^{(\mathrm{lr})}(\vec{r})$. The potential field $\phi^{(\mathrm{lr})}$ is the solution of the Poisson's equation (4.20). The charge density term is constructed as a sum of purely Gaussian functions and therefore is a smoothly varying function, furthermore, by construction the charge density is a periodic function with period L. As a result it is reasonable to assume that the corresponding potential field $\phi^{(\mathrm{lr})}$ is smoothly varying and periodic with period L. Hence a Fourier Transform approach is a suitable method to compute the form of the long-range potential $\phi^{(\mathrm{lr})}$.

We define the Fourier Transform of a function $f : \Omega \mapsto \mathbb{R}$ as

$$\hat{f}(\vec{k}) = \int_\Omega \exp(-i\vec{k} \cdot \vec{r}) f(\vec{r}) d\vec{r}, \tag{4.34}$$

and the Inverse Fourier Transform as

$$f(\vec{r}) = \frac{1}{V} \sum_{\vec{k}} \exp(i\vec{k} \cdot \vec{r}) \hat{f}(\vec{k}), \tag{4.35}$$

where V is the volume of the domain Ω. The vector $\vec{k}$ is a point in a reciprocal lattice defined by the simple cubic domain Ω. For a cuboid domain Ω formed by the three primitive vectors $(\vec{L}_1, \vec{L}_2, \vec{L}_3)$ we define the corresponding reciprocal lattice vectors

$$\vec{G}_1 = \frac{2\pi}{V}\vec{L}_2 \times \vec{L}_3, \tag{4.36}$$

$$\vec{G}_2 = \frac{2\pi}{V}\vec{L}_3 \times \vec{L}_1, \tag{4.37}$$

$$\vec{G}_3 = \frac{2\pi}{V}\vec{L}_1 \times \vec{L}_2. \tag{4.38}$$

A point $\vec{k}$ in the reciprocal lattice is given by a linear combination of the reciprocal lattice vectors with integer coefficients:

$$\vec{k} = g_1\vec{G}_1 + g_2\vec{G}_2 + g_3\vec{G}_3, \quad \text{where} \quad g_1, g_2, g_3 \in \mathbb{Z}. \tag{4.39}$$

The long range potential $\phi^{(\mathrm{lr})}$ is given by

$$-\Delta\phi^{(\mathrm{lr})}(\vec{r}) = 4\pi\rho^{(\mathrm{lr})}(\vec{r}), \tag{4.40}$$

where

$$\rho^{(\mathrm{lr})}(\vec{r}) = \sum_{\vec{\mathfrak{n}}\in\mathbb{Z}^3}\sum_{j=1}^{N} q_j S_\alpha(\vec{r} - \vec{r}_j - L\mathfrak{n}). \tag{4.41}$$

Applying the Fourier Transform to Equation (4.40) gives

$$|\vec{k}|^2\hat{\phi}^{(\mathrm{lr})}(\vec{k}) = 4\pi\hat{\rho}^{(\mathrm{lr})}(\vec{k}), \tag{4.42}$$

where

$$\hat{\rho}^{(\mathrm{lr})}(\vec{k}) = \int_\Omega \exp(-i\vec{k}\cdot\vec{r}) \sum_{\vec{\mathfrak{n}}\in\mathbb{Z}^3}\sum_{j=1}^{N} q_j S_\alpha(\vec{r} - \vec{r}_j - L\mathfrak{n})d\vec{r} \tag{4.43}$$

$$= \int_\Omega \exp(-i\vec{k}\cdot\vec{r}) \sum_{\vec{\mathfrak{n}}\in\mathbb{Z}^3}\sum_{j=1}^{N} q_j \left(\frac{\alpha}{\pi}\right)^{\frac{3}{2}} \exp(-\alpha|\vec{r} - \vec{r}_j - L\mathfrak{n}|^2)d\vec{r} \tag{4.44}$$

$$= \int_{\mathbb{R}^3} \exp(-i\vec{k}\cdot\vec{r}) \sum_{j=1}^{N} q_j \left(\frac{\alpha}{\pi}\right)^{\frac{3}{2}} \exp(-\alpha|\vec{r} - \vec{r}_j|^2)d\vec{r} \tag{4.45}$$

$$= \sum_{j=1}^{N} q_j \left(\frac{\alpha}{\pi}\right)^{\frac{3}{2}} \int_{\mathbb{R}^3} \exp(-i\vec{k}\cdot\vec{r}) \exp(-\alpha|\vec{r} - \vec{r}_j|^2)d\vec{r} \tag{4.46}$$

$$= \sum_{j=1}^{N} q_j \exp\left(\frac{-|\vec{k}|^2}{4\alpha}\right) \exp(-i\vec{k}\cdot\vec{r}_j). \tag{4.47}$$

We now apply the Inverse Fourier Transform to Equation (4.42) to give the long-range

contribution

$$\phi^{(\mathrm{lr})}(\vec{r}) = \frac{1}{V}\sum_{\vec{k}\neq\vec{0}} \exp(i\vec{k}\cdot\vec{r})\frac{4\pi}{|\vec{k}|^2}\sum_{j=1}^{N} q_j \exp\left(\frac{-|\vec{k}|^2}{4\alpha}\right)\exp(-i\vec{k}\cdot\vec{r}_j), \tag{4.48}$$

the $\vec{k} = \vec{0}$ case can be excluded as we assume the simulation domain carries zero net charge.

The Fourier Transform approach gives a potential field $\phi^{(lr)}$ from a charge density constructed with a Gaussian shaped density at the site of each charge. Hence the potential field $\phi^{(lr)}(\vec{r}_i)$ at charge i with position $\vec{r}_i$ includes the potential induced by the Gaussian $q_j S_\alpha(\vec{r}-\vec{r}_i)$. When evaluating $\phi^{(\mathrm{lr})}(\vec{r}_i)$ at the position of charge i there is a contribution to the potential from the Gaussian charge density $q_j S_\alpha(\vec{r}-\vec{r}_i)$, as in Figure 4-4, which is referred to as the self interaction. The self interaction of a charge does not contribute to the force exerted on the charge as the gradient is zero at centre of the Gaussian, but the self interaction does contribute to the potential energy. Hence the potential energy of the self interaction must be computed and subtracted from the total electrostatic potential energy.

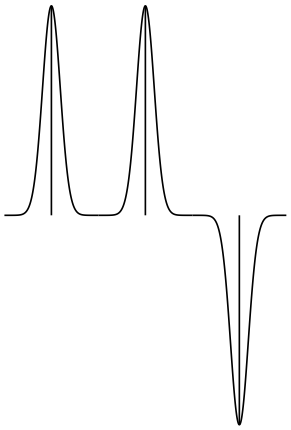

Figure 4-4: *One dimension representation of the self interaction between charges and their corresponding long-range charge density.*

The self interaction for each charge is the potential induced by a Gaussian charge density evaluated at the centre of the charge density. From Equation (4.27) the self interaction potential $\phi_j^{(\mathrm{self})}$ for charge q_j is

$$\phi_j^{(\mathrm{self})} = \left[\frac{2q_j}{r}\left(\frac{\alpha}{\pi}\right)^{\frac{1}{2}}\int_0^r \exp(-\alpha r'^2)dr'\right]\Bigg|_{r=0}, \tag{4.49}$$

$$= 2q_j\left(\frac{\alpha}{\pi}\right)^{\frac{1}{2}} \quad \text{using the mean-value theorem.} \tag{4.50}$$

Using $\phi_j^{(\text{self})}$ the self interaction energy for a system of N charges is given by

$$U^{(\text{self})} = \frac{1}{2}\sum_{j=1}^{N} q_j \phi^{(\text{self})}(\vec{r}_j), \tag{4.51}$$

$$= \left(\frac{\alpha}{\pi}\right)^{\frac{1}{2}} \sum_{j=1}^{N} q_j^2. \tag{4.52}$$

As the self interaction energy is independent of the position of the charge this quantity can be computed once at the beginning of the simulation and reused for a given set of constant charges. The total system energy is given by

$$U_{\text{coul}} = -U^{(\text{self})} + \frac{1}{2}\sum_{j=1}^{N}\left(q_j \phi^{(\text{lr})}(\vec{r}_j)\right) + \frac{1}{2}\sum_{j=1}^{N}\left(q_j \sum_{i\neq j} \phi_i^{(\text{sr})}(\vec{r}_j)\right), \tag{4.53}$$

where the i index in the second summation indexes particles in the primary image and surrounding periodic images. In Section 4.1.1 we argued that the Coulomb potential could not be truncated, in Appendix A.9 we investigate the convergence behaviour of the long-range potential for dipole charge distributions.

The electrostatic force exerted per unit charge is determined by the electric field, which is given by the spatial derivative of the long-range and short-range potential,

$$\vec{E}(\vec{r}) = -\vec{\nabla}\phi(\vec{r}) \tag{4.54}$$

$$= -\vec{\nabla}_{\vec{r}}\phi^{(\text{sr})}(\vec{r}) - \vec{\nabla}_{\vec{r}}\phi^{(\text{lr})}(\vec{r}). \tag{4.55}$$

If a charge q_j is positioned at the origin then the induced short-range force field $\vec{E}_j^{(\text{sr})}$ per unit charge at a point $\vec{r}$ is

$$\vec{E}_j^{(\text{sr})}(\vec{r}) = -\vec{\nabla}\phi_j^{(\text{sr})}(\vec{r}) \tag{4.56}$$

$$= -\frac{\partial}{\partial r}\phi_j^{(\text{sr})}(\vec{r})\frac{\vec{r}}{r} \tag{4.57}$$

$$= q_j\left(\frac{\text{erfc}(\sqrt{\alpha}r)}{r^2} + \frac{2\sqrt{\alpha}}{\sqrt{\pi}r}\exp(-\alpha r^2)\right)\frac{\vec{r}}{r}, \tag{4.58}$$

where $r = |\vec{r}|$. The short-range electric field $\vec{E}_j^{(\text{sr})}(\vec{r})$ converges to zero at an exponential rate, given by $\sqrt{\alpha}$, as the inter-particle distance r increases. Hence we truncate $\vec{E}_j^{(\text{sr})}(\vec{r})$ to zero at a radius $r_c \propto \alpha^{-1/2}$ to allow the short-range electric field $\vec{E}^{(\text{sr})}(\vec{r})$ to be computed. The short-range force field $\vec{E}^{\text{sr}}$ at a point $\vec{r}$ is given by the superposition of the short-range force fields induced by charged particles within a radius r_c of $\vec{r}$,

$$\vec{E}^{\text{sr}}(\vec{r}) = \sum_{j \text{ s.t. } |\vec{r}-\vec{r}_j|<r_c} \vec{E}_j^{\text{sr}}(\vec{r}_j - \vec{r}). \tag{4.59}$$

The long-range force field per unit charge $\vec{E}^{(\mathrm{lr})}$ is readily computed by taking the gradient of Equation (4.48),

$$\vec{E}^{(\mathrm{lr})}(\vec{r}) = -\vec{\nabla}_{\vec{r}}\,\phi^{(\mathrm{lr})}(\vec{r}), \tag{4.60}$$

$$= -\frac{1}{V}\sum_{\substack{\vec{k}\neq\vec{0}\\ |\vec{k}|<k_c}} i\vec{k}\exp(i\vec{k}\cdot\vec{r}_j)\frac{4\pi}{|\vec{k}|^2}\sum_{j=1}^{N} q_j \exp\left(\frac{-|\vec{k}|^2}{4\alpha}\right)\exp(-i\vec{k}\cdot\vec{r}_j), \tag{4.61}$$

where k_c is a cutoff for the maximum Fourier mode. No correction needs to be made to the long-range field $\vec{E}^{(\mathrm{lr})}$ for charge self interaction as the gradient at the centre of the Gaussian is zero.

4.2.1 Parameter Selection

An implementation of the method we have described contains three free parameters, the first of which is the width α of the Gaussian function used to split the charge density into a short-range and long-range components. The second parameter is the inter-particle distance r_c at which the short-range potential $\phi^{(\mathrm{sr})}$ is truncated to zero. The final parameter is the reciprocal space cutoff k_c which defines a maximum frequency to consider in the Fourier Transform and Inverse Fourier Transform used in the calculation of $\phi^{(\mathrm{lr})}$. Intuitively, as the Gaussian splitting function becomes narrower (i.e. α increases) we have to consider higher frequency modes in Fourier space (k_c increases) and are allowed to truncate the short-range potential at a shorter radius (r_c decreases).

Given an estimate of error induced by the real space cutoff r_c and reciprocal space cutoff k_c from Kolafa and Perram [45], Frenkel and Smit [24] derive a optimal parameter selection approach based on a computational cost model[1]. Kolafa and Perram derive the following expressions for the standard deviation of the real space error δE_R and standard deviation of the reciprocal space error δE_F,

$$\delta E_R = \left(\frac{r_c}{2L^3}\right)^{\frac{1}{2}}\frac{1}{\alpha r_c^2}\exp(-\alpha r_c^2)Q, \tag{4.62}$$

$$\delta E_F = \frac{k_c^{\frac{1}{2}}}{\sqrt{\alpha}L^2}\frac{1}{\left(\frac{\pi k_c}{\sqrt{\alpha}L}\right)^2}\exp\left(-\left(\frac{\pi k_c}{\sqrt{\alpha}L}\right)^2\right)Q, \tag{4.63}$$

where

$$Q = \sum_{j=1}^{N} q_j^2. \tag{4.64}$$

[1] [24] takes results from [45] without making a required $\alpha \to \sqrt{\alpha}$ substitution, here we make the substitution.

The standard deviations of both error estimates can be rewritten as

$$\delta E_R = \left(\frac{s}{2\sqrt{\alpha}L^3}\right)^{\frac{1}{2}} \frac{1}{s^2}\exp(-s^2)Q, \tag{4.65}$$

$$\delta E_F = \left(\frac{s}{\pi\sqrt{\alpha}L^3}\right)^{\frac{1}{2}} \frac{1}{s^2}\exp(-s^2)Q, \tag{4.66}$$

where we have made the choice:

$$r_c = \frac{s}{\sqrt{\alpha}}, \tag{4.67}$$

$$k_c = \frac{s\sqrt{\alpha}L}{\pi}, \tag{4.68}$$

for some constant s. These two error estimates are strongly influenced by the functional form $\exp(-s^2)/s^2$ such that a choice of s affects both errors in the same manner.

To control the error we choose some error tolerance ϵ and solve for s such that $\epsilon = \exp(-s^2)/s^2$. The remaining free parameter in Equations (4.67, 4.68) is α which is chosen to minimise the computational cost. We assume there exists a uniform distribution of N charges in a cube of side length L, hence the particle density in the simulation is N/L^3. For each charge i the short-range potential $q_i\phi_j^{(\mathrm{sr})}(\vec{r}_i)$ is evaluated for all charges j such that $|\vec{r}_i - \vec{r}_j| < r_c$, i.e. all neighbours within a sphere of radius r_c. With a particle density of N/L^3 the number of neighbours per particle is expected to be

$$N_{r_c} = \frac{4}{3}\frac{\pi r_c^3 N}{L^3}. \tag{4.69}$$

Hence if an evaluation of $\phi^{(\mathrm{sr})}$ has cost τ_R then the total cost of evaluating the short-range potential is approximately

$$C_R(\alpha) = \frac{4}{3}\pi\frac{N^2 s^3}{\alpha^{\frac{3}{2}}L^3}\tau_R. \tag{4.70}$$

For a given charge j and frequency $\vec{k}$ we denote the combined cost of evaluating $\exp\left(i\vec{k}\cdot\vec{r}_j\right)$ and $\exp\left(-i\vec{k}\cdot\vec{r}_j\right)$ as τ_F. We compute the Fourier Transform for all frequencies $\vec{k}$ such that $|\vec{k}| < k_c$ which defines a sphere in reciprocal space. Hence we estimate the cost of evaluating the long-range potential C_F by considering all vectors $\vec{k}$ within the sphere of radius k_c:

$$C_F(\alpha) = N\frac{4}{3}\pi k_c^3\tau_F \tag{4.71}$$

$$= \frac{4}{3\pi^2}s^3\alpha^{\frac{3}{2}}L^3 N\tau_F. \tag{4.72}$$

The total cost of the method is estimated by summing the two component costs,

$$C_R(\alpha) + C_F(\alpha) = \frac{4}{3}\pi\frac{N^2 s^3}{\alpha^{\frac{3}{2}}L^3}\tau_R + \frac{4}{3\pi^2}s^3\alpha^{\frac{3}{2}}L^3 N\tau_F. \tag{4.73}$$

Frenkel and Smit [26] find the minimum of Equation (4.73) by setting the derivative with respect to α to zero, this yields an optimal Gaussian width of

$$\alpha = \left(\frac{\tau_R \pi^3 N}{\tau_F L^6} \right)^{\frac{1}{3}}. \tag{4.74}$$

With the optimal choice of α the estimated cost model given by Equation (4.73) yields a computational cost of

$$C_R + C_F = \frac{4}{3\pi}(\tau_R + \tau_F) N^{\frac{3}{2}} \tag{4.75}$$

which is $\mathcal{O}(N^{\frac{3}{2}})$. If α is chosen independently of N the dominant term in the cost model is $L^3 N$ which leads to a $\mathcal{O}(N^2)$ computational complexity if $L \propto N^{\frac{1}{3}}$.

The optimal value of α is dependent on both the extent of the simulation domain and the number of charged particles and hence should ideally be computed for each simulation. Furthermore, computing an optimal α depends on accurate estimates of τ_R and τ_F which are expected to be machine dependent.

A variety of methods [18, 16, 22] exist that replace the Fourier Transform with a Fast Fourier Transform (FFT). As the FFT is typically only applicable to a regular grid these methods need to pay particular attention to the method used to interpolate functions to and from the grid. The benefit of a FFT approach is to reduce the calculation of the long-range potential to $\mathcal{O}(N \log N)$ complexity. These FFT accelerated methods are highly popular and implementations can be found in codes such as DL_POLY [41] and LAMMPS [63]. We do not consider these FFT based methods as many libraries already exist and instead describe the Fast Multipole Method (FMM) which in theory is computationally optimal with a complexity of $\mathcal{O}(N)$.

4.3 Fast Multipole Method

The Fast Multipole Method [31, 30, 32] is a hierarchical method to compute long-range interactions with a computational complexity that is linear in the number of charged particles. The main idea is to approximate the potential induced by a group of clustered together charges by an expansion which is valid far away from the charges. The cost to accuracy ratio of the method can be tuned by choosing the number of terms in the approximating expansion. We describe in detail the 2D version of the algorithm as described in [31] as the structure of the approach is identical to the 3D version and give a working overview of the 3D version.

4.3.1 Two Dimensional Fast Multipole Method

Multipole Expansion

We begin by representing a point $\vec{r} = (x, y) \in \mathbb{R}^2$ as the complex valued point $x + iy = z \in \mathbb{C}$. In 2D the fundamental solution to Poisson's Equation is logarithmic, if a unit charge

exists at a point z_0 then the induced potential field at a different point z is

$$\varphi(z) = \mathrm{Re}(-\log(z - z_0)). \tag{4.76}$$

In this derivation we write φ as the result of a complex valued function, the potential itself should be taken as the real part only. If a point z is such that $|z| > |z_0|$ then

$$\log(z - z_0) = \log(z) + \log\left(1 - \frac{z_0}{z}\right), \tag{4.77}$$

$$= \log(z) - \sum_{k=1}^{\infty} \frac{1}{k}\left(\frac{z_0}{z}\right)^k, \quad \text{as } \left|\frac{z_0}{z}\right| < 1. \tag{4.78}$$

Suppose that m charges of magnitudes $\{q_i, i = 1, \ldots, m\}$ are located at positions $\{z_i, i = 1, \ldots, m\}$ then by the superposition principle the induced potential at z is given by

$$\varphi(z) = \sum_{i=1}^{m} -q_i \log(z - z_i). \tag{4.79}$$

If we assume that the m charges are clustered around the origin such that $|z_i| < r$ for $i = 1, \ldots, m$ then at a point z where $|z| > r$ we can write $\varphi(z)$ as an expansion centred at the origin by using Equations (4.77, 4.78),

$$\varphi(z) = \sum_{i=1}^{m} [-q_i \log(z)] + \sum_{i=1}^{m} \left[q_i \sum_{k=1}^{\infty} \frac{1}{k}\left(\frac{z_i}{z}\right)^k \right] \tag{4.80}$$

$$= \sum_{i=1}^{m} [-q_i \log(z)] + \sum_{k=1}^{\infty} \frac{1}{z^k} \left[\sum_{i=1}^{m} \frac{q_i z_i^k}{k} \right] \tag{4.81}$$

$$= a_0 \log(z) + \sum_{k=1}^{\infty} \frac{a_k}{z^k} \tag{4.82}$$

where

$$a_k = \begin{cases} \sum_{i=1}^{m} -q_i & \text{if } k = 0 \\ \sum_{i=1}^{m} \frac{q_i z_i^k}{k} & \text{if } k > 0. \end{cases} \tag{4.83}$$

Equation (4.82) is referred to as a multipole expansion. The computational attraction of the multipole expansion stems from the independence of the expansion coefficients a_k from the evaluation point z. The expansion coefficients can be computed once for a set of charges then be subsequently used to compute the potential at any sufficiently far away point.

In practice the expansion coefficients a_k can only be computed for some finite range of k, we denote the index of the maximum computed coefficient by p. A truncated multipole expansion φ_p gives an approximation to the true potential φ with an error given by the

non-computed terms

$$|\varphi_p - \varphi| = \left| \varphi(z) - a_0 \log(z) - \sum_{k=1}^{p} \frac{a_k}{z^k} \right| \tag{4.84}$$

$$= \left| \sum_{k=p+1}^{\infty} \frac{a_k}{z^k} \right|. \tag{4.85}$$

Using the definition of a_k we have

$$|\varphi_p - \varphi| \leq A \sum_{k=p+1}^{\infty} \frac{r^k}{k|z|^k} \tag{4.86}$$

$$\leq A \sum_{k=p+1}^{\infty} \left|\frac{r}{z}\right|^k = \alpha \left|\frac{r}{z}\right|^{p+1} = \left(\frac{A}{c-1}\right)\left(\frac{1}{c}\right)^p \tag{4.87}$$

where

$$A = \sum_{i=1}^{m} |q_i|, \quad \alpha = \frac{A}{1 - |r/z|} \quad \text{and} \quad c = \left|\frac{z}{r}\right|. \tag{4.88}$$

If we consider a set of charges contained in a square domain that has been subdivided into 16 square cells, as in Figure 4-5, then each charge is contained within a parent cell. At the centre of each cell a p-term multipole expansion is computed from the charges contained within that cell. The error bound in Equation (4.86-4.87) indicates that a multipole expansion computed about the centre of a cell cannot be evaluated with any reasonable accuracy in any of the adjacent cells. For example, the multipole expansion computed at the cell centre P in Figure 4-5 will not give an accurate approximation of the potential φ in the shaded region, but it will give a good approximation in the unshaded cells. We refer to cells separated by more than one cell as well separated.

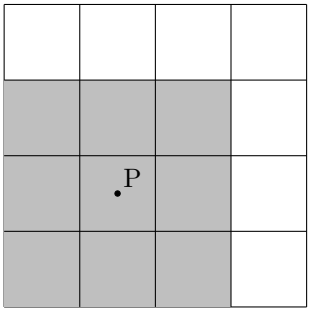

Figure 4-5: *A p-term multipole expansion at P constructed from charges contained in the square containing P will not give an accurate approximation of the potential φ in the shaded region.*

The multipole expansion alone can be used as a more efficient method to compute the potential between well separated clusters of charges. Consider two well separated cells with centres P and Q that contain N and M charges respectively as in Figure 4-6. Naively computing the potential at each particle site has a computational complexity that is $\mathcal{O}(NM)$ which can be reduced to $\mathcal{O}(N) + \mathcal{O}(M)$ by using multipole expansions.

First a p-term multipole expansion is constructed at the points P and Q from the charges contained in each cell at the cost of $\mathcal{O}(N) + \mathcal{O}(M)$ work. The p-term expansion at the point P can be thought of as an approximation to the charge density in the containing cell and can be evaluated at each of the M points in the second cell to give an approximation to the potential at each point. Similarly, the expansion centred at Q is evaluated at each of the N points in the first cell. The total cost of evaluating potentials from expansions is $\mathcal{O}(N)+\mathcal{O}(M)$ which yields a reduced computational complexity in comparison to $\mathcal{O}(NM)$ at the cost of reduced accuracy.

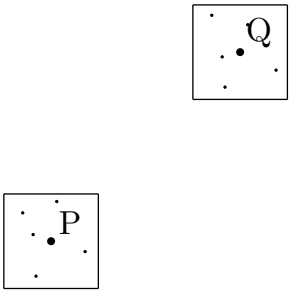

Figure 4-6: *Two well separated cells P and Q.*

Multipole to Multipole Translation

The main idea to reach a computational complexity of $\mathcal{O}(N)$ is to group together increasingly larger clusters of particles in a hierarchical approach. If two p-term expansions share an origin then, by using linearity, a single p-term expansion can be formed from the element-wise sum of terms in the two expansions. If four p-term expansions are centred in adjacent cells, as in Figure 4-7, then they do not share an origin and cannot be immediately combined through addition. First the origin of each multipole expansion must be translated to the intersection of the four cells such that they can be combined into a single multipole expansion, this operation is referred to as a multipole-to-multipole translation.

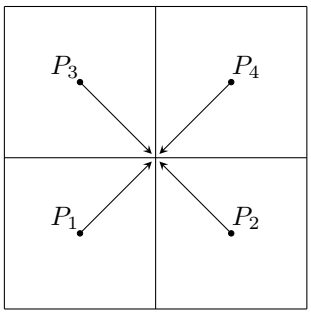

Figure 4-7: *Four p-term multipole expansions at the points $P_1, \ldots, P_4$ are translated to the intersection point of the four cells.*

Suppose that

$$\varphi(z) = a_0 \log(z - z_0) + \sum_{k=1}^{\infty} \frac{a_k}{(z - z_0)^k}, \tag{4.89}$$

is a multipole expansion that describes the potential induced by m charges $q_1, q_2, \ldots, q_m$ clustered within a circle of radius R centred at z_0. Then this multipole expansion can be shifted to the origin such that for a point z where $|z| > R + |z_0|$

$$\varphi(z) = a_0 \log(z) + \sum_{l=1}^{\infty} \frac{b_l}{z^l}, \tag{4.90}$$

where

$$b_l = \left(\sum_{k=1}^{l} a_k z_0^{l-k} \binom{l-1}{k-1} \right) - \frac{a_0 z_0^l}{l}. \tag{4.91}$$

The error in the potential φ from a p-term expansion is given by

$$\left| \varphi(z) - a_0 \log(z) - \sum_{l=1}^{p} \frac{b_l}{z^l} \right| \leq \left(\frac{A}{1 - \left| \frac{|z_0|+R}{z} \right|} \right) \left| \frac{|z_0| + R}{z} \right|^{p+1}, \tag{4.92}$$

where

$$A = \sum_{i=1}^{m} |q_i| \quad \text{and} \quad |z| > |z_0| + R. \tag{4.93}$$

The proof uses the following expansion as shown in Appendix A.2,

$$(z - z_0)^{-k} = \sum_{l=k}^{\infty} \binom{l-1}{k-1} z_0^{l-k} z^{-l}, \tag{4.94}$$

and the identity,

$$\sum_{l=1}^{\infty} \sum_{k=1}^{\infty} a_k \binom{l-1}{k-1} \frac{z_0^{l-k}}{z^l} = \sum_{l=1}^{\infty} \sum_{k=1}^{l} a_k \binom{l-1}{k-1} \frac{z_0^{l-k}}{z^l}. \tag{4.95}$$

Given the expansion in Equation (4.94) and the identity in Equation (4.95) we rewrite Equation (4.89) as

$$\varphi(z) = a_0 \log(z) - a_0 \sum_{l_1=1}^{\infty} \frac{1}{l_1} \left(\frac{z_0}{z} \right)^{l_1} + \sum_{k=1}^{\infty} a_k \sum_{l_2=k}^{\infty} \binom{l_2-1}{k-1} z_0^{l_2-k} z^{-l_2}, \tag{4.96}$$

$$= a_0 \log(z) + \sum_{l=1}^{\infty} \left[-\frac{z_0^l a_0}{l z^l} + \sum_{k=1}^{l} a_k \binom{l-1}{k-1} \frac{z_0^{l-k}}{z^l} \right], \tag{4.97}$$

$$= a_0 \log(z) + \sum_{l=1}^{\infty} \frac{b_l}{z^l}, \tag{4.98}$$

where

$$b_l = \sum_{k=1}^{l} a_k \binom{l-1}{k-1} z_0^{l-k} - \frac{z_0^l a_0}{l}. \tag{4.99}$$

In the transformation of Equation (4.96) into Equation (4.97) Greengard has assumed that

the summations are absolutely convergent and hence can be reordered. Hence a p-term multipole expansion can be translated to the origin with a computational complexity of $\mathcal{O}(p^2)$. Furthermore, after translation, Equation (4.92) states that the new radius where the expansion is invalid is larger and includes the original circle where the expansion was invalid.

Multipole to Local Translation

While the multipole expansion is accurate far away from the centre of the expansion, a local expansion is accurate near the centre of the expansion. Multipole expansions are utilised as descriptors of charge densities and local expansions are used to describe potential fields in the regions near the centre of the expansions. Suppose that m charges are clustered within a circle of radius R and centre z_0 such that $|z_0| > (c+1)R$ where $c > 1$. Then the multipole expansion constructed from these m charges converges in a circle of radius R centred at the origin. Inside the circle centred at the origin the potential due to the m charges can be described by the power series

$$\varphi(z) = \sum_{l=0}^{\infty} b_l \cdot z^l, \tag{4.100}$$

where

$$b_0 = \sum_{k=1}^{\infty} \frac{a_k}{z_0^k}(-1)^k + a_0 \log(-z_0), \tag{4.101}$$

and

$$b_l = \left(\frac{1}{z_0^l} \sum_{k=1}^{\infty} \frac{a_k}{z_0^k} \binom{l+k-1}{k-1} (-1)^k \right) - \frac{a_0}{l \cdot z_0^l} \quad \text{for } l \geq 1. \tag{4.102}$$

The error of a p-term expansion where $p \geq \max(2, 2c/(c-1))$ is given by

$$\left| \varphi(z) - \sum_{l=0}^{p} b_l \cdot z^l \right| < \frac{\left(\sum_{i=1}^{m} |q_i|\right) \left(4e(p+c)(c+1) + c^2\right)}{c(c-1)} \left(\frac{1}{c}\right)^{p+1}, \tag{4.103}$$

where e is the base of the natural logarithm. This error bound indicates that a multipole expansion cannot be converted into a local expansion near to the original origin of expansion without significant loss of accuracy. Furthermore, for the 2D FMM algorithm Equation (4.103) gives a method by which to choose the required number of expansion terms p based on the required output accuracy.

We refer to a power series such as in Equation (4.100) as a local expansion as it is valid in the region near the centre of the expansion. To obtain the form of the local expansion in Equations (4.101 - 4.102) compute the Maclaurin series of Equation (4.89). The l-th derivative at zero of Equation (4.89) is given by

$$\left. \frac{\partial^l \varphi(z)}{\partial z^l} \right|_{z=0} = -(l-1)! \frac{a_0}{z_0^l} + \sum_{k=1}^{\infty} (-1)^k \frac{(k+l-1)!}{(k-1)!} \frac{a_k}{z_0^{k+l}}. \tag{4.104}$$

Hence the local expansion of φ is

$$\varphi(z) = a_0 \log(-z_0) + \sum_{k=1}^{\infty} \frac{a_k}{(-z_0)^k} + \sum_{l=1}^{\infty} \frac{z^l}{l!} \left[\sum_{k=1}^{\infty} (-1)^k \frac{(k+l-1)!}{(k-1)!} \frac{a_k}{z_0^{k+l}} - (l-1)! \frac{a_0}{z_0^l} \right], \tag{4.105}$$

$$= a_0 \log(-z_0) + \sum_{k=1}^{\infty} \frac{a_k}{(-z_0)^k} + \sum_{l=1}^{\infty} z^l \left[\sum_{k=1}^{\infty} (-1)^k \binom{l+k-1}{k-1} \frac{a_k}{z_0^{k+l}} - \frac{a_0}{z_0^l l} \right]. \tag{4.106}$$

Like multipole expansions, if two local expansions share a centre of expansion then the terms from both can be combined by element-wise addition to form a single expansion. Converting a multipole expansion into a local expansion has a computational complexity $\mathcal{O}(p^2)$.

Local to Local Translation

The origin of p-term local expansion can be shifted without loss of precision. This allows two or more expansions to be combined by first shifting the origins to a common location then adding the terms. If

$$\varphi(z) = \sum_{k=0}^{p} a_k z^l, \tag{4.107}$$

is a local expansion then by the Binomial Theorem

$$\varphi(z + z_0) = \sum_{k=0}^{p} a_k \sum_{l=0}^{k} \binom{k}{l} z_0^{k-l} z^l, \tag{4.108}$$

$$= \sum_{l=0}^{p} \left(\sum_{k=l}^{p} a_k \binom{k}{l} z_0^{k-l} \right) z^l. \tag{4.109}$$

The process of shifting the origin of a local expansion is referred to as a local to local translation and has a computational complexity of $\mathcal{O}(p^2)$. From a physical perspective, combining two local expansions is equivalent to combining two potential fields through the superposition principle.

The Fast Multipole Algorithm

We give a description of the 2D algorithm from Greengard and Rokhlin [31]. The method applies a hierarchical approach to provide a computational complexity of $\mathcal{O}(N)$. In this algorithm the number of terms in all expansions is fixed at a value $p \approx \log_2(\epsilon)$ where ϵ is an error tolerance. We assume that N charges are contained in a unit square on which a hierarchy of $\mathcal{L}$ meshes are defined. Mesh level $l \in \{0, \ldots, \mathcal{L} - 1\}$ subdivides the unit square into 4^l equally sized squares indexed by $\{0, \ldots, 4^l - 1\}$ as illustrated in Figure 4-8. In this decomposition, each square on a level l is subdivided into 4 squares on level $l + 1$ which are referred to as the children of the larger parent square.

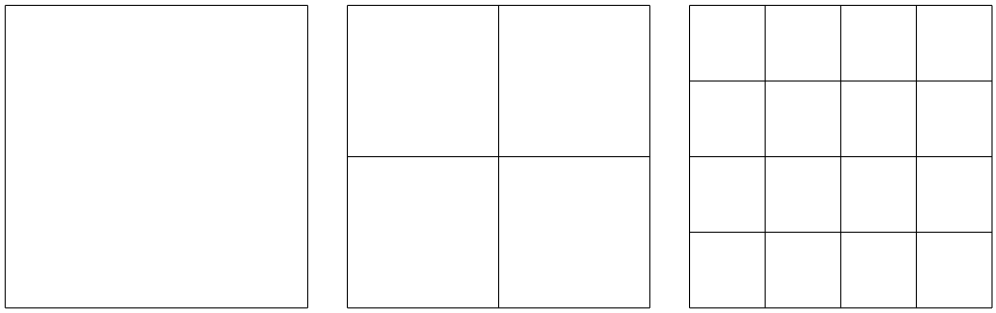

Figure 4-8: *The hierarchy of mesh levels for* $\mathcal{L} = 3$.

We define following notation to describe the algorithm,

$\Phi_{l,i}$: The p-term multipole expansion about the centre of square i on level l that describes the potential induced by charges within square i.

$\Psi_{l,i}$: The p-term local expansion at the centre of square i on level l that describes the potential induced by all charges outside the square i and its 8 nearest neighbours.

$\bar{\Psi}_{l,i}$: The p-term expansion about the centre of square i on level l that describes the potential induced by all charges outside the parent of the square and outside the 8 nearest neighbours of this parent.

interaction list: The interaction list for a square i on level l contains squares on level l which are the children of the parent square of i and its nearest neighbours which are well separated from square i, as in Figure 4-9.

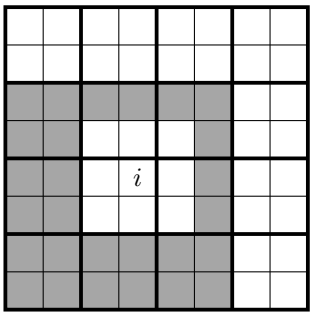

Figure 4-9: *Interaction list in grey for square* i. *Thick lines indicate the boundaries of the parent cells.*

Algorithms 18 and 19 give a description of the 2D algorithm applicable to a system with free space boundary conditions. More complex boundary conditions, such as periodic boundary conditions, are discussed in Section 4.3.1. The resulting algorithm has a computational complexity $\mathcal{O}(N)$ and an overview of the cost of each step is given in Table 4-10.

Step	Cost	Explanation
Multipole construction	$\mathcal{O}(Np)$	Each of the N charges contributes to one p-term multipole expansion.
Upward pass	$\mathcal{O}(Np^2)$	Multipole to multipole translation has cost $\mathcal{O}(p^2)$ and there are $\mathcal{O}(N)$ cells on the finest level. The multipole expansion in each of the $\mathcal{O}(N)$ cells on the finest is translated to the origin of the parent cell.
Downward pass - local to local	$\mathcal{O}(Np^2)$	Local to local translation has cost $\mathcal{O}(p^2)$ and there are $\mathcal{O}(N)$ cells on the finest level. Each of the $\mathcal{O}(N)$ cells on the finest level is the target of a single local to local translation.
Downward pass - multipole to local	$\mathcal{O}(Np^2)$	A multipole to local translation has cost $\mathcal{O}(p^2)$ and there are $\mathcal{O}(N)$ cells on the finest level. Furthermore, the interaction list for each cell has at most 27 entries on any level.
Direct interactions	$\mathcal{O}(N)$	Each of the N charges directly interacts with charges in the 9 surrounding cells where each cell contains $\mathcal{O}(1)$ charges.

Figure 4-10: *Overview of 2D FMM cost per step, adapted from [31].*

Algorithm 18: 2D FMM algorithm to compute $\Psi_{l,i}$ for a system with free space boundary conditions.

Input: N charges, a maximum level of mesh refinement $\mathcal{L}$ and a number of terms p.
Output: $\Psi_{l,i}$

Upward pass:

Form the p-term multipole expansions at the centre of each square on the finest level $\mathcal{L}-1$.

```
for i = 0, ..., 4^{L-1} - 1 do
    Construct Φ_{L-1,i}
end
```

Form the p-term multipole expansion on each coarser level by using multipole to multipole translation to shift the child expansions to the centre of the parent square.

```
for l = L - 1, ..., 0 do
    for i = 0, ..., 4^l - 1 do
        Construct Φ_{l,i} from
        Φ_{l+1,children(i)}
    end
end
```

Downward pass:

By traversing from coarsest mesh to finest mesh the local expansions $\Psi_{l,i}$ are formed in each square on each mesh level.

For $l \in \{0, \ldots, \mathcal{L}-1\}$ do:

On a layer l translate the local expansion of each parent on layer $l-1$ to the centres of the 4 child squares

```
for i = 0, ..., 4^l - 1 do
    Construct Ψ̄_{l,i} from the local to
    local translation of Ψ_{l-1,parent(i)}
end
```

Construct $\Psi_{l,i}$ though multipole to local (MTL) translation of expansions in the interaction list (IL) of square i

```
for i = 0, ..., 4^l - 1 do
    Ψ_{l,i} ← Ψ̄_{l,i}
    for j ∈ IL(i) do
        Ψ_{l,i} ← Ψ_{l,i} + MTL(Φ_{l,j})
    end
end
```

In the upward pass multipole expansions are computed on levels 0 and 1, however, with free space boundary conditions there are no well separated squares on these two mesh levels. In this scenario, the downward pass is initialised by setting $\Psi_{0,0} = \bar{\Psi}_{0,0} = \Psi_{1,i} = \bar{\Psi}_{1,i} = 0$ for all i. Given the local expansions $\Psi_{\mathcal{L}-1,i}$ the potential energy can be computed for each charge using Algorithm 19. A pass of Algorithms 18 and 19 is illustrated in Figure 4-11 which shows the propagation of information to and from a single

square on the finest mesh level.

Algorithm 19: 2D FMM algorithm to compute the interactions between charges via p-term local expansions and direct charge-charge interactions.

Input: N charges and local expansions $\Psi_{\mathcal{L}-1,i}$.
Output: Potential energies and optionally forces.

At the end of the downward pass the expansions $\Psi_{\mathcal{L}-1,i}$ have been computed. To compute the potential energy and force of each charge the expansions $\Psi_{\mathcal{L}-1,i}$ are used in conjunction with a local direct calculation,

For a charge in square i the expansion $\Psi_{\mathcal{L}-1,i}$ approximates the potential field from all squares well separated from i on level $\mathcal{L}-1$.

```
foreach square i = 0, ..., 4^(L-1) − 1 do
    foreach charge k in i do
        Compute interactions with well
        separated charges via Ψ_{L-1,i}
    end
end
```

For a given charge k in square i the interactions with other charges in square i and in the 8 nearest neighbour squares of square i are computed directly.

```
foreach square i = 0, ..., 4^(L-1) − 1 do
    foreach charge k in i do
        Compute interactions with
        charges in i and the nearest
        neighbours of i directly
    end
end
```

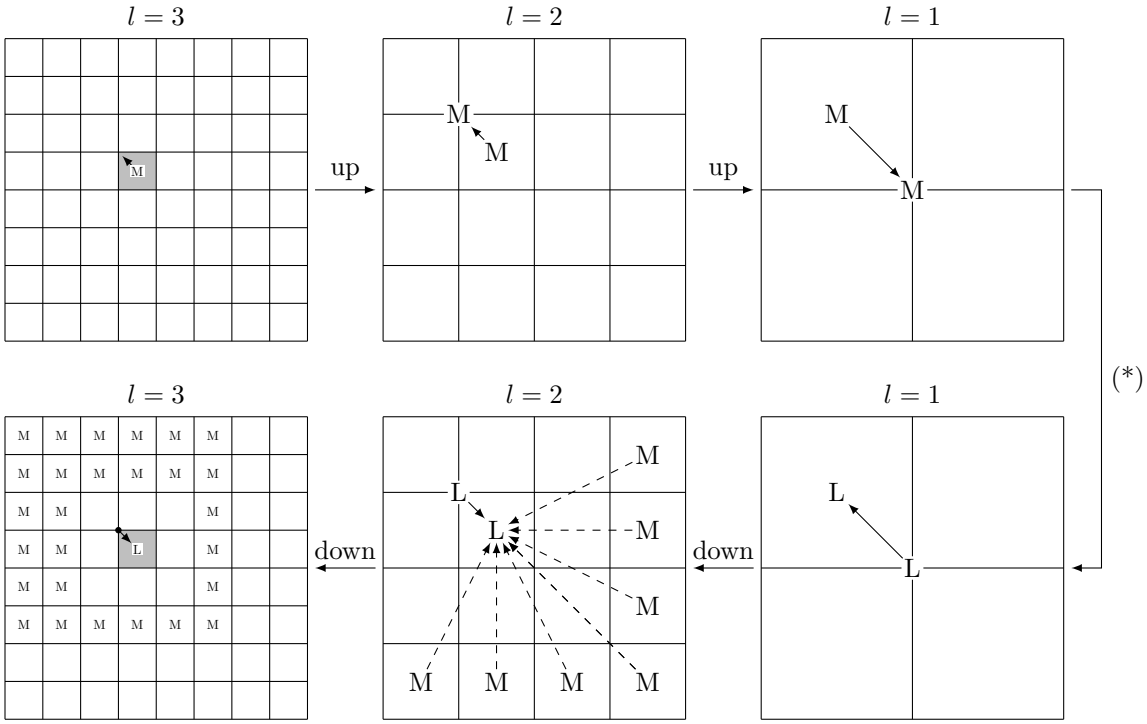

Figure 4-11: *Overview of the flow of information to and from the grey square in a pass of the 2D FMM. Multipole to Multipole and Local to Local translations are illustrated with solid line arrows. Multipole to Local translations are denoted by dashed line arrows. For clarity multipole to local arrows are omitted on mesh level 3; the multipole source locations are indicated by "M" and the centre of local expansions by "L". The arrow denoted by "(*)" indicates a generic boundary condition method on level 0 that converts the p-term expansion $\Phi_{0,0}$ to $\Psi_{0,0}$ if applicable.*

Free Space and Periodic Boundary Conditions

The simplest boundary conditions to consider are free space boundary conditions where the simulation domain is surrounded by an infinite vacuum. Free space boundaries can be implemented in the 2D and 3D method by setting $\Psi_{0,0} = \bar{\Psi}_{0,0} = \Psi_{1,i} = \bar{\Psi}_{1,i} = 0$ and truncating interaction lists at the boundaries of the domain.

Simulations often require periodic boundary conditions, the potential at a point z in the primary image due to all periodic images is given by

$$\varphi^p(z) = \sum_{\mathfrak{n} \neq \vec{0}} \varphi_{\mathfrak{n}}(z), \tag{4.110}$$

where $\mathfrak{n} \in \mathbb{Z}^2$ indexes periodic images and $\varphi_{\mathfrak{n}}(z)$ is the potential induced by periodic image $\mathfrak{n}$ at the point z. In Figure 4-12 each square is an image of the simulation domain and the centre grey square is the primary image. At the end of the upward pass of the FMM the multipole expansion $\Phi_{0,0}$ approximates the potential induced by the primary image and is valid in well separated images. Hence the copies of $\Phi_{0,0}$ centred in the hashed region of Figure 4-12 can be evaluated in the primary image and copies in the white region cannot be evaluated in the primary image.

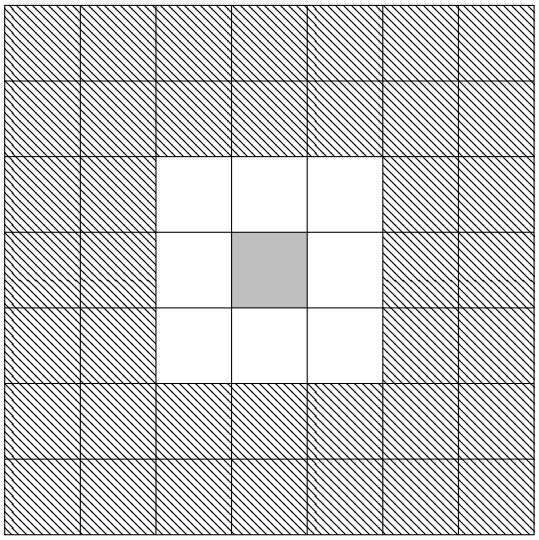

Figure 4-12: *Simulation domain and periodic images. Central grey image represents the simulation domain, all other squares represent periodic images. The multipole expansion $\Phi_{0,0}$ centred in squares in the white region is not valid in the primary image.*

We split the potential induced by periodic images into two components, one from periodic images that are well separated from the primary image and hence the multipole expansion $\Phi_{0,0}$ can be evaluated, and a second component containing the adjacent images:

$$\varphi^p(z) = \sum_{|\mathfrak{n}|_\infty > 1} \varphi_{\mathfrak{n}}(z) + \sum_{|\mathfrak{n}|_\infty = 1} \varphi_{\mathfrak{n}}(z), \tag{4.111}$$

where $|\mathfrak{n}|_\infty$ denotes the infinity norm of $\mathfrak{n}$. The first term in Equation (4.111) refers to the diagonally hatched region of Figure 4-12 and the second term refers to the white region. The second summation over the adjacent squares is readily computed in practice by allowing interaction lists used in the downwards pass of the algorithm to pass over the boundary and index squares in periodic images. Secondly, in the direct charge to charge stage of the downward pass charges which reside in squares on the edge of the domain must consider interactions with charges over the periodic boundary.

For example, with free space boundary conditions all squares on mesh level $l = 1$ have empty interaction lists as there are no well separated squares. However, with periodic boundary conditions all squares on mesh level $l = 1$ have "complete" interaction lists that only involve squares in the periodic images. Figure 4-13 illustrates the primary image (in grey) surrounded by the adjacent periodic images (in white). The crosses in the figure indicate the required interaction list required to compute $\Psi_{1,P}$ under the assumption that $\bar{\Psi}_{1,P}$ contains the contribution from all well separated periodic images.

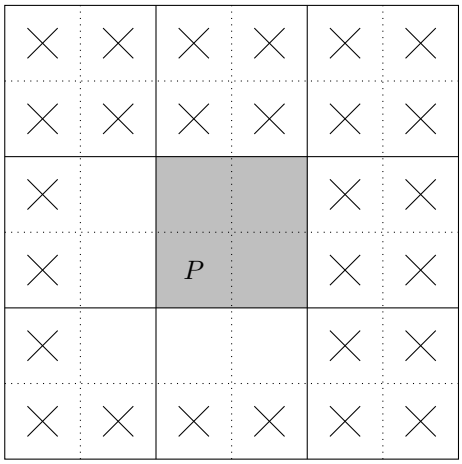

Figure 4-13: *Interaction list marked with crosses for square P. The grey square indicates the primary image and white squares indicate periodic images. Dotted lines indicate boundaries between the four squares per image.*

The contribution to $\varphi^p(z)$ from all well separated periodic images is $\Psi_{0,0}$. We assume that the total charge a_0 in the domain is zero otherwise $\Psi_{0,0}$ will be infinite. The multipole expansion at the centre of each well separated periodic image is translated to a local expansion at the centre of the primary image by

$$\Psi_{0,0} = \sum_{|\mathfrak{n}|_\infty>1} \left(\sum_{k=1}^{\infty} \frac{a_k}{(-z_\mathfrak{n})^k} + \sum_{l=1}^{p} z^l \left[\sum_{k=1}^{\infty} (-1)^k \binom{l+k-1}{k-1} \frac{a_k}{z_\mathfrak{n}^{k+l}} \right] \right), \tag{4.112}$$

$$= \sum_{k=1}^{\infty} a_k (-1)^k \left(\sum_{|\mathfrak{n}|_\infty>1} \frac{1}{z_\mathfrak{n}^k} \right) + \sum_{l=1}^{p} z^l \left[\sum_{k=1}^{\infty} (-1)^k a_k \binom{l+k-1}{k-1} \sum_{|\mathfrak{n}|_\infty>1} \left(\frac{1}{z_\mathfrak{n}^{k+l}} \right) \right], \tag{4.113}$$

where $z_\mathfrak{n} \in \mathbb{C}$ is the centre of the periodic image $\mathfrak{n}$. This summation over the infinite lattice of periodic images is not initially well defined as terms of the form

$$\sum_{|\mathfrak{n}|_\infty>1} \frac{1}{z_\mathfrak{n}^m}, \tag{4.114}$$

are conditionally convergent for $m < 3$, for $m \geq 3$ the sum can be precomputed and stored.

Greengard and Rokhlin [31] propose a solution that ensures the summation is convergent in 2D by imposing further physically realistic restrictions. For the $m = 1$ case Greengard and Rokhlin consider a system with a single unit charge in the centre of the box, in the periodic system each charge has a net force of zero due to Newton's Third Law. The force on the charge in the primary image is given by

$$\sum_{\mathfrak{n}} \left(-\frac{d}{dz} \log(z) \Big|_{z=z_\mathfrak{n}} \right) = \sum_{\mathfrak{n}} \frac{-1}{z_\mathfrak{n}}. \tag{4.115}$$

Hence if we assert that the net force on the charge in the primary is zero then we deduce

$$\sum_{\mathrm{n}} \frac{-1}{z_{\mathrm{n}}} = 0 = \sum_{\mathrm{n}} \frac{1}{z_{\mathrm{n}}}. \tag{4.116}$$

Hence in the two summations in Equation (4.113) that are conditionally convergent the $m = 1$ case is set to be zero. Greengard and Rokhlin make a similar argument with a infinite lattice of dipoles and deduce that for the $m = 2$ case

$$\sum_{\mathrm{n}} \frac{1}{z_{\mathrm{n}}^2} = \pi. \tag{4.117}$$

We discuss the summation in 3D in section 4.3.2. In summary periodic boundary conditions are realised in 2D by computing $\Psi_{0,0}$ via a summation over all periodic images and on mesh level l the values of $\Psi_{l,i}$ are computed by allowing interaction lists to index squares in periodic images.

4.3.2 Three Dimensional Fast Multipole Method

This section provides a working overview of the 3D fast multipole method described by Greengard [32, 30]. The 3D FMM algorithm is very similar in structure to the 2D algorithm, many components are directly replaced by their 3D counterparts. For brevity proofs are omitted but are given by Greengard in [30]. As the fundamental solution to Poisson's Equation has a different functional form in 3D from 2D, different expansions are required to describe the 3D potential fields. This description of the algorithm is written in spherical coordinates where a point P in space is described by a tuple (r, θ, ϕ) where $r \in [0, \infty)$ is the radial distance to the origin, $\theta \in [0, \pi]$ is the polar angle measured from the z-axis and $\phi \in [0, 2\pi)$ is the azimuthal angle measured from the x-axis as in Figure 4-14.

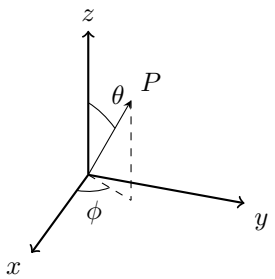

Figure 4-14: *Spherical coordinate convention.*

As discussed in Section 4.1, the potential φ at a point $P = (r, \theta, \phi)$ induced by a unit charge at the point $P_0 = (\rho, \alpha, \beta)$ is

$$\varphi(P) = \frac{1}{R}, \tag{4.118}$$

where R is the separation distance between the two points. The 3D FMM writes the $1/R$ potential as an infinite sum of terms, each term is a product of a P dependent function and a P_0 dependent function. If γ is the angle between the vectors P and P_0 then by the cosine rule

$$R^2 = r^2 + \rho^2 - 2r\rho\cos(\gamma). \tag{4.119}$$

Hence

$$\frac{1}{R} = \frac{1}{r\sqrt{1 - 2u\mu + \mu^2}}, \tag{4.120}$$

where

$$u = \cos(\gamma), \quad u = \frac{\rho}{r}. \tag{4.121}$$

For $\mu < 1$ the inverse square root in Equation (4.120) can be written as the series

$$\frac{1}{\sqrt{1 - 2u\mu + \mu^2}} = \sum_{n=0}^{\infty} P_n(u)\mu^n, \tag{4.122}$$

where $P_n(u)$ is the Legendre Polynomial of degree n. Computationally, the expansion in Equation (4.122) is not particularly useful as both terms in the summation couple the source and destination points.

We use the spherical harmonics to obtain an expansion where summation terms depend on either the source or destination point. The spherical harmonics are solutions to the angular component of the Laplace equation in spherical coordinates. Any harmonic function φ can be written as an expansion around the origin

$$\varphi(r, \theta, \phi) = \sum_{n=0}^{\infty} \sum_{m=-n}^{n} L_n^m r^n Y_n^m(\theta, \phi), \tag{4.123}$$

where L_n^m are known as the local moments. This expansion is referred to as a local expansion as when the expansion is truncated at $n = p - 1$ the resulting p-term expansion is accurate near to the centre of expansion. A harmonic function φ can also be written as a multipole expansion which when truncated is valid far away from the centre of expansion,

$$\varphi(r, \theta, \phi) = \sum_{n=0}^{\infty} \sum_{m=-n}^{n} \frac{M_n^m}{r^{n+1}} Y_n^m(\theta, \phi), \tag{4.124}$$

where M_n^m are known as multipole moments of the expansion. We provide the error bounds for multipole and local expansions in later sections.

The exact definition of the spherical harmonics $Y_n^m(\theta, \phi)$ varies between physics textbook sources, we use the definition

$$Y_n^m(\theta, \phi) = \sqrt{\frac{(n - |m|)!}{(n + |m|)!}} P_n^{|m|}(\cos(\theta)) \exp(im\phi), \tag{4.125}$$

where P_n^m is the $(n, m)^{\text{th}}$ associated Legendre polynomial and $i = \sqrt{-1}$.

The following theorem from [30, 32] relates the Legendre polynomials to the spherical harmonics and allows Equation (4.122) to be rewritten in a computationally more useful manner.

Addition theorem for Legendre polynomials

Consider the spherical coordinate points $P = (r, \theta, \phi)$ and $Q = (\rho, \alpha, \beta)$ and let γ be the angle subtended between them. Then

$$P_n(\cos(\gamma)) = \sum_{m=-n}^{n} Y_n^{-m}(\alpha, \beta) Y_n^m(\theta, \phi). \tag{4.126}$$

Through direct substitution Equation (4.120) is rewritten as

$$\frac{1}{R} = \sum_{n=0}^{\infty} \sum_{m=-n}^{n} \rho^n Y_n^{-m}(\alpha, \beta) \frac{Y_n^m(\theta, \phi)}{r^{n+1}}, \tag{4.127}$$

which is computationally more useful as summation terms can be grouped into those dependent on the source point and those dependent on the evaluation point. In the 3D FMM this identity allows the charge density of a cluster of charges to be approximated at some central point by an expansion. The expansion can then be evaluated repeatedly at far away points to give the potential field from the cluster of charges.

Multipole expansion

Here we describe the 3D variant of the 2D multipole expansion in Equation (4.82). Suppose that l charges $\{q_i, i = 1, \ldots, l\}$ are located at positions $\{Q_i = (\rho_i, \alpha_i, \beta_i), i = 1, \ldots, l\}$ such that $|\rho_i| < a \forall i$. Then at a point $P = (r, \theta, \phi)$ where $r > a$ the potential $\varphi(P)$ is given by

$$\varphi(P) = \sum_{n=0}^{\infty} \sum_{m=-n}^{n} \frac{M_n^m}{r^{n+1}} Y_n^m(\theta, \phi), \tag{4.128}$$

where

$$M_n^m = \sum_{i=1}^{l} q_i \rho_i^n Y_n^{-m}(\alpha_i, \beta_i). \tag{4.129}$$

If the infinite sum is truncated at $p \geq 1$ terms then an error bound is given by

$$\left| \varphi(P) - \sum_{n=0}^{p} \sum_{m=-n}^{n} \frac{M_n^m}{r^{n+1}} Y_n^m(\theta, \phi) \right| \leq \frac{1}{r - a} \left(\frac{a}{r} \right)^{p+1} \sum_{i=1}^{l} |q_i|. \tag{4.130}$$

The coefficients $M_n^m \in \mathbb{C}^{(p+1)^2}$ describe the potential induced by the l charges at a point P that is well separated from the origin, the centre of this multipole expansion is at the origin. The following operation translates an expansion centred around the point Q to an expansion centred around the origin at the expense of increasing the radius at which

it can be accurately evaluated.

Translation of a multipole expansion

As in the 2D FMM method (Equations (4.90) and (4.91)), multipole-to-multipole translation allows several multipole expansions to be combined by shifting the centres of the expansions to a common origin, the coefficients can then simply be added together. However, the radius of the sphere in which the new expansion is invalid increases to the sum of the original radius plus the distance the expansion was translated through.

Suppose that l charges $\{q_i, i = 1, \ldots, l\}$ are located within a sphere D centred at $Q = (\rho, \alpha, \beta)$ with radius a, and that at a point $P = (r, \theta, \phi)$ outside D, the potential due to these charges is given by the expansion

$$\varphi(P) = \sum_{n=0}^{\infty} \sum_{m=-n}^{n} \frac{O_n^m}{r'^{n+1}} Y_n^m(\theta', \phi'), \tag{4.131}$$

where $P - Q = (r', \theta', \phi')$. Then the expansion can be translated to the origin such that for a point $P = (r, \theta, \phi)$ outside the sphere of radius $a + \rho$ centred at the origin,

$$\varphi(P) = \sum_{j=0}^{\infty} \sum_{k=-j}^{j} \frac{M_j^k}{r^{j+1}} Y_j^k(\theta, \phi), \tag{4.132}$$

where

$$M_j^k = \sum_{n=0}^{j} \sum_{m=-n}^{n} \frac{1}{A_j^k} O_{j-n}^{k-m} i^{|k|-|m|-|k-m|} A_n^m A_{j-n}^{k-m} \rho^n Y_n^{-m}(\alpha, \beta), \tag{4.133}$$

and

$$A_n^m = \frac{(-1)^n}{\sqrt{(n-m)!(n+m)!}}. \tag{4.134}$$

Furthermore, the error in the potential at the point P from a p-term expansion is bounded by

$$\left| \varphi(P) - \sum_{j=0}^{p} \sum_{k=-j}^{j} \frac{M_j^k}{r^{j+1}} Y_j^k(\theta, \phi) \right| \leq \left(\frac{\sum_{i=1}^{l} |q_i|}{r - (a + \rho)} \right) \left(\frac{a + \rho}{r} \right)^{p+1} \tag{4.135}$$

Multipole to local conversion

This conversion and translation of a multipole expansion to a local expansion is identical in process to the 2D variant in Equations (4.100, 4.101). Suppose that l charges $\{q_i, i = 1, \ldots, l\}$ are located within a sphere D centred at $Q = (\rho, \alpha, \beta)$ with radius a, and that $\rho > (c + 1)a$ with $c > 1$. Then the multipole expansion in Equation (4.132) converges inside a sphere D_0 of radius a centred at the origin. Inside D_0 the potential induced by

charges $\{q_i, i = 1, \ldots, l\}$ is described by the local expansion

$$\varphi(P) = \sum_{j=0}^{\infty} \sum_{k=-j}^{j} L_j^k Y_j^k(\theta, \phi) r^j, \tag{4.136}$$

where,

$$L_j^k = \sum_{n=0}^{\infty} \sum_{m=-n}^{n} \frac{O_n^m i^{|k-m|-|k|-|m|} A_n^m A_j^k Y_{j+n}^{m-k}(\alpha, \beta)}{(-1)^n A_{j+n}^{m-k} \rho^{j+n+1}}. \tag{4.137}$$

Furthermore, for $p \geq 1$,

$$\left| \varphi(P) - \sum_{j=0}^{p} \sum_{k=-j}^{j} L_j^k Y_j^k(\theta, \phi) r^j \right| \leq \left(\frac{\sum_{i=1}^{l} |q_i|}{a(c-1)} \right) \left(\frac{1}{c} \right)^{p+1} \tag{4.138}$$

Translation of a local expansion

The following result in Equation (4.141) provides the machinery to shift the centre of a local expansion similar to the 2D case described in Equations (4.107, 4.109). As in the 2D case, if two local expansions share a centre of expansion then a single local expansion may be formed from the sum of the coefficients. Let $Q = (\rho, \alpha, \beta)$ be the origin of the p-term local expansion

$$\varphi(P) = \sum_{n=0}^{p} \sum_{m=-n}^{n} O_n^m Y_n^m(\theta', \phi') r'^j, \tag{4.139}$$

where $P = (r, \theta, \phi)$ and $P - Q = (r', \theta', \phi')$. Then

$$\varphi(P) = \sum_{j=0}^{p} \sum_{k=-j}^{j} L_j^k Y_j^k(\theta, \phi) r^j, \tag{4.140}$$

where

$$L_j^k = \sum_{n=j}^{p} \sum_{m=-n}^{n} \frac{O_n^m i^{|m|-|m-k|-|k|} A_{n-j}^{m-k} A_j^k Y_{n-j}^{m-k}(\alpha, \beta) \rho^{n-j}}{(-1)^{n+j} A_n^m}. \tag{4.141}$$

The structure of the 3D FMM algorithm is identical to the 2D variant. In the 2D case a hierarchy of mesh levels was placed on the simulation domain where mesh level l subdivided the domain into 4^l identical squares. For a 3D simulation domain the squares are replaced with cubes such that a mesh level l subdivides the domain into 8^l identical cubes. For convenience the following notation is defined:

$\Phi_{l,i}$ the p-term multipole expansion about the centre of cube i on level l that describes the potential induced by charges within cube i.

$\Psi_{l,i}$ the p-term local expansion at the centre of cube i on level l that describes the potential induced by all charges outside the cube i and its 26 nearest neighbours.

$\bar{\Psi}_{l,i}$ the p-term expansion about the centre of cube i on level l that describes the potential

induced by all charges outside the parent of the cube and outside the 26 nearest neighbours of this parent.

$\mathcal{T}_{\mathrm{MM}}$ the linear operator mapping the multipole moments $\{O_j^k : j \in [0,p], k \in [-j,j]\}$ to the multipole moments $\{M_j^k : j \in [0,p], k \in [-j,j]\}$ using Equation (4.133).

$\mathcal{T}_{\mathrm{ML}}$ the linear operator mapping the multipole moments $\{O_j^k : j \in [0,p], k \in [-j,j]\}$ to the local moments $\{L_j^k : j \in [0,p], k \in [-j,j]\}$ using Equation (4.137).

$\mathcal{T}_{\mathrm{LL}}$ the linear operator mapping the local moments $\{O_j^k : j \in [0,p], k \in [-j,j]\}$ to the local moments $\{L_j^k : j \in [0,p], k \in [-j,j]\}$ using Equation (4.141).

interaction list (IL): The interaction list for a cube i on level l contains cubes on level l which are the children of the parent cube of i and its nearest neighbours which are well separated from cube i. In general, this list contains 189 entries, the 2D variant is illustrated in Figure 4-9.

A pass of the algorithm is initialised by choosing a value of p, the number of expansion terms, based on the desired accuracy. For an algorithm that exhibits a computational complexity $\mathcal{O}(N)$ the number of mesh levels is chosen as $\mathcal{L} \approx \log_8(N)$.

Algorithm 20: 3D FMM algorithm to compute $\Psi_{l,i}$ for a system with free space boundary conditions.

Input: N charges, a maximum level of mesh refinement $\mathcal{L}$ and a number of terms p.
Output: $\Psi_{l,i}$

Upward pass:

Form the p-term multipole expansions at the centre of each cube on the finest level.

```
for i = 0, ..., 8^(L-1) - 1 do
    Construct Φ_{L-1,i}
end
```

Form the p-term multipole expansion on each coarser level by using $\mathcal{T}_{\mathrm{MM}}$ to shift the child expansions to the centre of the parent cube.

```
for l = L - 1, ..., 0 do
    for i = 0, ..., 8^l - 1 do
        Φ_{l,i} = 0⃗
        for k ∈ children(i) do
            Φ_{l,i} ← Φ_{l,i} + T_MM(Φ_{l+1,k})
        end
    end
end
```

Downward pass:

By traversing from coarsest mesh to finest mesh the local expansions $\Psi_{l,i}$ are formed in each square on each mesh level.

For $l \in \{0, \ldots, \mathcal{L}-1\}$ do:

Translate each local expansion $\Psi_{l-1,i}$ on layer $l-1$ to the centres of the 8 child cubes on layer l using the $\mathcal{T}_{LL}$ operator.

```
for i = 0, ..., 8^l - 1 do
    Ψ̄_{l,i} ← T_LL(Ψ_{l-1,parent(i)})
end
```

Construct $\Psi_{l,i}$ though multipole to local translation of expansions in the interaction list (IL) of cube i

```
for i = 0, ..., 8^l - 1 do
    Ψ_{l,i} ← Ψ̄_{l,i}
    for j ∈ IL(i) do
        Ψ_{l,i} ← Ψ_{l,i} + T_ML(Φ_{l,j})
    end
end
```

As in the 2D version, in the upward pass multipole expansions are computed on levels 0 and 1, with free space boundary conditions there are no well separated cubes on these two mesh levels thus these two levels can be neglected in this scenario. Hence with free space boundary conditions $\Psi_{0,0} = \bar{\Psi}_{0,0} = \Psi_{1,i} = \bar{\Psi}_{1,i} = 0$ for all i.

Given the local expansions $\Psi_{\mathcal{L}-1,i}$ the potential energy and force can be computed for each charge using Algorithm 21. The electric field is given by the gradient of the potential field and is required to compute forces, Appendix A.4 derives the equations to compute the electric field from a local expansion.

Algorithm 21: 3D FMM algorithm to compute the interactions between charges via p-term local expansions and direct charge-charge interactions.

Input: N charges and local expansions $\Psi_{\mathcal{L}-1,i}$.
Output: Potential energies and optionally forces.

At the end of the downward pass the expansions $\Psi_{\mathcal{L}-1,i}$ have been computed. To compute the potential energy and force of each charge the expansions $\Psi_{\mathcal{L}-1,i}$ are used in conjunction with a local direct calculation,

For a charge in cube i the expansion $\Psi_{\mathcal{L}-1,i}$ approximates the potential field from all cubes well separated from i on level $\mathcal{L}-1$.

```
foreach cube i = 0, ..., 8^{L-1} - 1 do
    foreach charge k in i do
        Compute interactions with well separated charges via Ψ_{L-1,i}
    end
end
```

For a given charge k in cube i the interactions with other charges in cube i and in the 26 nearest neighbour cubes of cube i are computed directly.

```
foreach cube i = 0, ..., 8^{L-1} - 1 do
    foreach charge k in i do
        Compute interactions with charges in i and the nearest neighbours of i directly
    end
end
```

Computational complexity

If the number of levels $\mathcal{L}$ is approximately $\log_8(N)$ then the average number of charges per cube on the finest level s is $\mathcal{O}(1)$. Furthermore, the total number of cubes N_c is $\mathcal{O}(N)$ as

$$N_c = \sum_{r=0}^{\mathcal{L}-1} 8^r = \frac{1}{7}\left(8^{\mathcal{L}} - 1\right) = \frac{1}{7}\left(8^{\log_8(N)} - 1\right) = \mathcal{O}(N). \tag{4.142}$$

The computational complexity of each stage of the algorithm is as follows:

- Upward pass

1. Construction of multipole expansions from charges: $\mathcal{O}(Np^2)$, each particle contributes to one multipole expansion consisting of p^2 coefficients.
2. Multipole to multipole translation: $\mathcal{O}(Np^4)$, there are N_c cubes which perform one multipole to multipole translation via the $\mathcal{T}_{\mathrm{MM}}$ operator which has computational complexity $\mathcal{O}(p^4)$.

- Downward pass

1. Local to local translation: $\mathcal{O}(Np^4)$, there are N_c cubes which perform one local to local translation via the $\mathcal{T}_{\mathrm{LL}}$ operator which has computational complexity $\mathcal{O}(p^4)$.
2. Multipole to local translation: $\mathcal{O}(Np^4)$, there are N_c cubes which perform 189 multipole to local translations via the $\mathcal{T}_{\mathrm{ML}}$ operator which has computational complexity $\mathcal{O}(p^4)$. This step forms the majority of the computational work which manipulates multipole and local expansions.
3. Interaction with the "well separated" field though local expansions: $\mathcal{O}(Np^2)$, the interaction between each charge and other charges in well separated cubes occurs through the local expansion which consists of p^2 terms.
4. Direct charge-charge interactions: $\mathcal{O}(N)$, each charge interacts directly with charges in the cube it resides in and the 26 neighbouring cubes, each cube contains $\mathcal{O}(1)$ charges by construction.

Hence the overall computational complexity of the method is $\mathcal{O}(N)$. Although this is linear in the number of particles the coefficient is highly non-trivial and is governed by the $\mathcal{O}(p^4)$ complexity of the translation operators. In particular the computational cost of the multipole to local operations is the most significant.

Periodic boundary conditions

As in the 2D FMM, particular attention is given to the computation of $\Psi_{0,0}$ with periodic boundary conditions. Computing $\Psi_{0,0}$ requires the summation over the infinite lattice of periodic images excluding the primary image and its nearest neighbours. More formally, given expansion coefficients O_n^m such that

$$\Phi_{0,0} = \sum_{n=0}^{p} \sum_{m=-n}^{n} \frac{O_n^m}{r^{n+1}} Y_n^m(\theta, \phi), \tag{4.143}$$

the objective is to compute coefficients L_j^k such that

$$\Psi_{0,0} = \sum_{j=0}^{p} \sum_{k=-j}^{j} L_j^k r^j Y_j^k(\theta, \phi). \tag{4.144}$$

We consider the lattice of well separated periodic images, as illustrated for the 2D case in Figure 4-12, and define $\vec{r}_{\vec{\nu}} = (\rho_{\vec{\nu}}, \alpha_{\vec{\nu}}, \beta_{\vec{\nu}})$ to be the vector to the centre of a well separated image $\vec{\nu} \in \{\vec{x} \in \mathbb{Z}^3 : |\vec{x}|_\infty > 1\}$. By applying the multipole to local operator $\mathcal{T}_{\text{ML}}$ to each well separated periodic image we determine that the coefficients of the local expansion $\Psi_{0,0}$ are given by

$$L_j^k = \sum_{\vec{\nu}} \mathcal{T}_{\text{ML}}\left(O_n^m, \vec{r}_{\vec{\nu}}\right), \tag{4.145}$$

$$= \sum_{\vec{\nu}} \left(\sum_{n=0}^{p} \sum_{m=-n}^{n} \frac{O_n^m i^{|k-m|-|k|-|m|} A_n^m A_j^k Y_{j+n}^{m-k}(\alpha_{\vec{\nu}}, \beta_{\vec{\nu}})}{(-1)^n A_{j+n}^{m-k} \rho_{\vec{\nu}}^{j+n+1}} \right), \tag{4.146}$$

as can be deduced from the definition of $\mathcal{T}_{\text{ML}}$ and Equation (4.137). As the images all share the same multipole coefficients the multipole coefficients factor out of the summation over periodic images,

$$L_j^k = \sum_{n=0}^{p} \sum_{m=-n}^{n} \frac{O_n^m i^{|k-m|-|k|-|m|} A_n^m A_j^k}{(-1)^n A_{j+n}^{m-k}} \sum_{\vec{\nu}} \left(\frac{Y_{j+n}^{m-k}(\alpha_{\vec{\nu}}, \beta_{\vec{\nu}})}{\rho_{\vec{\nu}}^{j+n+1}} \right), \tag{4.147}$$

$$= \sum_{n=0}^{p} \sum_{m=-n}^{n} \frac{O_n^m i^{|k-m|-|k|-|m|} A_n^m A_j^k}{(-1)^n A_{j+n}^{m-k}} R_{j+n}^{m-k}, \tag{4.148}$$

where

$$R_n^m = \sum_{\vec{\nu}} \frac{Y_n^m(\alpha_{\vec{\nu}}, \beta_{\vec{\nu}})}{\rho_{\vec{\nu}}^{n+1}}. \tag{4.149}$$

We provide an overview of the method by Amisaki [7] which applies an Ewald derived approach to compute the matrix R_n^m for a system that exhibits no net charge and zero dipole moment. Although this process is potentially expensive, once the matrix R is computed it can be reused until the extent of the simulation domain is changed. The method uses the gamma function $\Gamma(z)$, incomplete gamma function $\gamma(a, x)$ and complementary gamma function $\Gamma(a, x)$ defined as

$$\Gamma(z) = \int_0^\infty t^{z-1} \exp(-t) dt = \lambda^z \int_0^\infty t^{z-1} \exp(-\lambda t) dt. \tag{4.150}$$

$$\gamma(a, x) = \int_0^x t^{a-1} \exp(-t) dt, \tag{4.151}$$

$$\Gamma(a, x) = \int_x^\infty t^{a-1} \exp(-t) dt. \tag{4.152}$$

Starting with the gamma function and the substitutions $\lambda = r_\nu^2$ and $z = n + 1/2$, the matrix R is written as

$$R_n^m(\vec{r}_{\vec{\nu}}) = F_n^m(\vec{r}_{\vec{\nu}}) + G_n^m(\vec{r}_{\vec{\nu}}), \tag{4.153}$$

where

$$F_n^m(\vec{r}_{\vec{\nu}}) = \frac{Y_n^m(\alpha_{\vec{\nu}}, \beta\vec{\nu})}{\Gamma(n+\frac{1}{2})} \int_0^{\kappa^2} r_{\vec{\nu}}^n t^{n-1/2} \exp(-r_{\vec{\nu}}^2 t) dt, \tag{4.154}$$

$$= \frac{Y_n^m(\alpha_{\vec{\nu}}, \beta\vec{\nu})\gamma(n+1/2, \kappa^2 r_{\vec{\nu}}^2)}{\Gamma(n+\frac{1}{2}) r_{\vec{\nu}}^{n+1}}, \tag{4.155}$$

$$G_n^m(\vec{r}_{\vec{\nu}}) = \frac{Y_n^m(\alpha_{\vec{\nu}}, \beta\vec{\nu})}{\Gamma(n+\frac{1}{2})} \int_{\kappa^2}^{\infty} r_{\vec{\nu}}^n t^{n-1/2} \exp(-r_{\vec{\nu}}^2 t) dt, \tag{4.156}$$

$$= \frac{Y_n^m(\alpha_{\vec{\nu}}, \beta\vec{\nu})\Gamma(n+1/2, \kappa^2 r_{\vec{\nu}}^2)}{\Gamma(n+\frac{1}{2}) r_{\vec{\nu}}^{n+1}}. \tag{4.157}$$

The magnitude of $G_n^m(\vec{r}_{\vec{\nu}})$ decays rapidly as $r_{\vec{\nu}}$ increases and is analogous to the short-range component of the Ewald summation method, the matrix G_n^m is computed directly for periodic images within a certain distance of the primary image. The convergence of the summation in the computation of F_n^m is governed by the $1/r_{\vec{\nu}}^{n+1}$ term, this summation is computed in reciprocal space see [7] for details.

For a cuboid simulation cell Amisaki argues that by angular symmetry the matrix R is real valued and that the $(n,m)^{\text{th}}$ entry is zero if n or m are odd. Furthermore, if the simulation cell is cubic by using spherical symmetry Amisaki claims the $(n,m)^{\text{th}}$ term is zero if $m \neq 0 \pmod 4$ or if $n = 2$. For large values of r the summation $\sum_{\vec{\nu}} 1/r_{\vec{\nu}}^{n+1}$ can be estimated by the integrals in Section 4.1.1 where we demonstrated the integral cannot be truncated for $n < 3$. With these properties of the matrix R all summations of the form $\sum_{\vec{\nu}} 1/r_{\vec{\nu}}^{n+1}$ for $n < 3$ are chosen to be zero, hence all conditionally convergent summations have been assigned a physically sensible value.

For each n and m the corresponding entry R_n^m is computed as the addition of a summation over reciprocal space and summation in real space, finally the contribution from the nearest neighbours are explicitly subtracted,

$$\begin{aligned} R_n^m = & \\ & \sum_{\substack{\vec{h} \text{ s.t. } v_{\vec{h}} < h_c \\ \text{and } \vec{h} \neq \vec{0}}} \frac{i^n \pi^{n-1/2} Y_n^m(\alpha_{\vec{h}}, \beta_{\vec{h}}) v_{\vec{h}}^{n-2} \exp(-\pi^2 v_{\vec{h}}^2/\kappa^2)}{\Gamma(n+1/2)V} \\ & + \sum_{\substack{\vec{\nu} \in \mathbb{Z}^3 \\ \text{s.t. } \vec{\nu} \neq \vec{0},\ r_{\vec{\nu}} < r_c}} \frac{Y_n^m(\alpha_{\vec{\nu}}, \beta_{\vec{\nu}})\Gamma(n+1/2, \kappa^2 r_{\vec{\nu}}^2)}{\Gamma(n+1/2) r_{\vec{\nu}}^{n+1}} \\ & - \sum_{\substack{\vec{\nu} \in \mathbb{Z}^3 \\ \text{s.t. } |\vec{\nu}|_\infty = 1}} \frac{Y_n^m(\alpha_{\vec{\nu}}, \beta_{\vec{\nu}})}{r_{\vec{\nu}}^{n+1}}, \end{aligned} \tag{4.158}$$

where h_c is a maximum frequency to consider in reciprocal space and r_c is a cutoff in real space. The index $\vec{h} = (h_a, h_b, h_c) \in \mathbb{Z}^3$ denotes the vector $h_a\vec{a}^* + h_b\vec{b}^* + h_c\vec{c}^*$ in reciprocal space with corresponding spherical coordinates $(v_{\vec{h}}, \alpha_{\vec{h}}, \beta_{\vec{h}})$. The parameters

h_c, r_c and κ should be chosen such that the errors induced in $\Psi_{0,0}$ by the errors in the elements of R are negligible in comparison to the global error, see [7] for a discussion on parameter selection. From an implementation standpoint, the matrix R is precomputed once at the beginning of the simulation which induces a relatively tiny overall cost, hence it is preferable to be "pessimistic" in the parameter selection stage to ensure accuracy is not lost in the application of the matrix R.

Rotation Matrices

For a p-term expansion the multipole to local operation denoted by $\mathcal{T}_{\text{ML}}$ exhibits a $\mathcal{O}(p^4)$ computational complexity if performed with Equation (4.137). Greengard and Rokhlin [32] present the following approach to reduce the computational complexity to $\mathcal{O}(p^3)$ by using rotation matrices. Consider a translation vector parallel to the z axis: $(\rho, 0, 0)$, the multipole to local translation requires the evaluation of

$$Y_n^m(0,0) = \begin{cases} \sqrt{\frac{(n-|m|)!}{(n+|m|)!}} & \text{if } m = 0 \\ 0 & \text{otherwise} \end{cases}. \tag{4.159}$$

Hence for a p-term expansion Equation (4.137) reduces to

$$L_j^k = \sum_{n=0}^{p} \frac{O_n^k A_n^k A_j^k Y_{j+n}^0(0,0)}{(-1)^n A_{j+n}^0 \rho^{j+n+1}}, \tag{4.160}$$

which allows z-direction multipole to local translation with $\mathcal{O}(p^3)$ computational complexity as there are p^2 expansion coefficients indexed by j and k that each require p operations to translate. The operator $\mathcal{T}_{\text{ML}}^z(\rho)$ denotes the application of $\mathcal{T}_{\text{ML}}$ along the z-axis by distance ρ through applying Equation (4.160). The same approach can be applied to the multipole to multipole and local to local translations, however, these operations do not significantly contribute to the computation time and hence we focus on the multipole to local translation case.

To apply the $\mathcal{T}_{\text{ML}}^z$ operator we require machinery to rotate the coordinate systems of multipole expansions and local expansions. Assume the potential at a point $P = (r, \theta, \phi)$ is given by

$$\varphi(P) = \sum_{n=0}^{\infty} \sum_{m=-n}^{n} B_n^m Y_n^m(\theta, \phi), \tag{4.161}$$

where B_n^m refers to a local or multipole expansion at the origin:

$$B_n^m = \begin{cases} L_n^m r^n & \text{local expansion} \\ \frac{M_n^m}{r^{n+1}} & \text{multipole expansion} \end{cases} \tag{4.162}$$

If the coordinate system is rotated around the z-axis through an angle β then in the new

coordinate system $P = (r, \theta, \phi')$ and

$$\varphi(P) = \sum_{n=0}^{\infty} \sum_{m=-n}^{n} e^{im\beta} B_n^m Y_n^m(\theta, \phi'). \tag{4.163}$$

Rotation of multipole or local coefficients O_n^m around the z-axis through angle β is denoted by the operator $\mathcal{R}_z(\beta) : O_n^m \mapsto O_n^m \exp(im\beta)$. The operator $\mathcal{R}_z(\beta)$ is diagonal and hence exhibits a $\mathcal{O}(p^2)$ computational complexity.

The second rotation operation is a rotation of the coordinate system around the y-axis. Assuming the potential at the point $P = (r, \theta, \phi)$ is given by Equation (4.161) and the coordinate system is rotated around the y-axis through an angle α then in the new coordinate system $P = (r, \theta', \beta)$ and

$$\varphi(P) = \sum_{n=0}^{\infty} \sum_{m=-n}^{n} \bar{B}_n^m Y_n^m(\theta', \phi), \tag{4.164}$$

where

$$\bar{B}_n^m = \begin{cases} r^n \sum_{m'=-n}^{n} d_n^{m,m'}(\alpha) L_n^{m'} & \text{local expansion} \\ \frac{1}{r^{n+1}} \sum_{m'=-n}^{n} d_n^{m,m'}(\alpha) M_n^{m'} & \text{multipole expansion} \end{cases} \tag{4.165}$$

and $d_n^{m,m'}(\alpha)$ is the Wigner d-matrix [80, 28]. Rotation of coefficients around the y-axis through angle α is denoted by the operator $\mathcal{R}_y(\alpha) : O_n^m \mapsto \sum_{m'=-n}^{n} d_n^{m,m'} O_n^{m'}$ and has $\mathcal{O}(p^3)$ computational complexity. Combining the rotation operations and z-direction translation operation gives

$$\mathcal{T}_{\mathrm{ML}} = \mathcal{R}_z(-\beta)\mathcal{R}_y(-\alpha)\mathcal{T}_{\mathrm{ML}}^z(\rho)\mathcal{R}_y(\alpha)\mathcal{R}_z(\beta), \tag{4.166}$$

where (ρ, α, β) is the translation vector. By using Equation (4.166) the operators $\mathcal{R}_z$, $\mathcal{R}_y$ and $\mathcal{T}_{\mathrm{ML}}^z$ directly replace the operator $\mathcal{T}_{\mathrm{ML}}$ in Algorithm 20 without any further modification to the algorithm. An overview of this rotation matrix approach is illustrated in Figure 4-15.

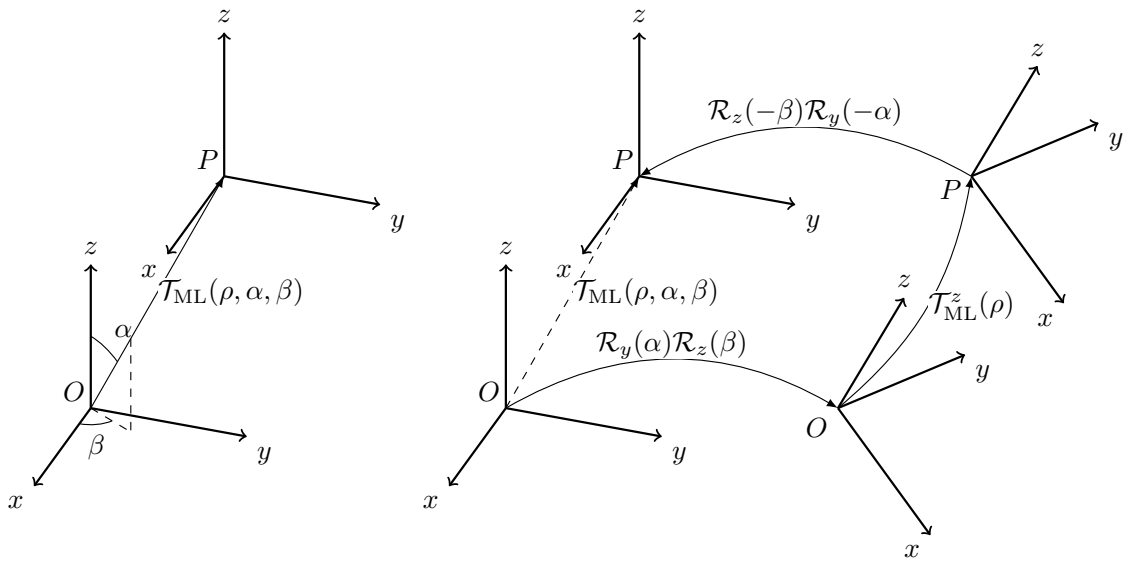

Figure 4-15: *Left: original multipole to local translation $\mathcal{T}_{ML}$ along the vector (ρ, α, β) with $\mathcal{O}(p^4)$ computational complexity. Right: multipole to local translation performed by (1) rotating coordinate frame with operation $\mathcal{R}_y(\alpha)\mathcal{R}_z(\beta)$ (2) z-direction multipole to local translation $\mathcal{T}^z_{ML}$ along new z-axis (3) rotate coefficients back into the original coordinate frame with operation $\mathcal{R}_z(-\beta)\mathcal{R}_y(-\alpha)$.*

If all translation operators were modified to use rotation matrices then the complexity of the 3D FMM would be reduced from $\mathcal{O}(Np^4)$ to $\mathcal{O}(Np^3)$. We only replace the multipole to local translation, hence our implementation is still asymptotically $\mathcal{O}(Np^4)$. In practice the $\mathcal{T}_{\mathrm{ML}}$ operator is applied ≈ 189 times more often than $\mathcal{T}_{\mathrm{MM}}$ or $\mathcal{T}_{\mathrm{LL}}$ hence when $p = \mathcal{O}(10)$ we see a significant improvement without using rotation matrices for these operators.

CHAPTER 5

IMPLEMENTATION OF ELECTROSTATIC INTERACTION ALGORITHMS

We now discuss our implementations of the Ewald summation method and the FMM. Sections 5.1 and 5.2 are published in the ParCo 2017 conference proceedings [74].

5.1 Ewald Implementation

We implemented the Ewald method within our abstraction and code generation framework described in Sections 2.2 and 3.1. In summary, the short-range component is implemented with a *Local Particle Pair Loop* and the long-range component is implemented with a pair of *Particle Loops* and one *GlobalArray*.

Short Range Potential

By construction, the short-range potential $\phi^{(\mathrm{sr})}(\vec{r})$ rapidly converges to zero as the inter-particle distance $|\vec{r}|$ increases. We truncate the short-range contribution to the electrostatic potential and force with a cutoff r_c (see Section 4.2.1),

$$\phi^{(\mathrm{sr})}_{r_c}(\vec{r}) = \sum_{\substack{j \text{ with} \\ |\vec{r}-\vec{r}_j|<r_c}} q_j \frac{\operatorname{erfc}\left(\sqrt{\alpha}|\vec{r}-\vec{r}_j|\right)}{|\vec{r}-\vec{r}_j|} \tag{5.1}$$

$$\vec{F}^{(\mathrm{sr})}_{r_c}(\vec{r}) = \sum_{\substack{j \text{ with} \\ |\vec{r}-\vec{r}_j|<r_c}} q_i q_j \frac{\vec{r}-\vec{r}_j}{|\vec{r}-\vec{r}_j|^2} \left[\frac{\operatorname{erfc}\left(\sqrt{\alpha}|\vec{r}-\vec{r}_j|\right)}{|\vec{r}-\vec{r}_j|} + 2\sqrt{\frac{\alpha}{\pi}} \exp(-\alpha|\vec{r}-\vec{r}_j|^2) \right].$$

The computational kernel for the *local* `ParticlePair` loop is given in Listing 5.1. The position and charge data are stored per particle in `ParticleDat` data objects. Similarly, the resulting forces and total potential energy are stored as a `ParticleDat` and a `GlobalArray` object. Listing 5.2 shows the corresponding Python code for launching the pair loop. In the C-kernel capitalised variables, such as `REAL_CUTOFF_SQ`, are constants

which are replaced by their numerical values at compile time using the `kernel_consts` dictionary.

Listing 5.1: *Implementation of the short range force in Equation (5.1) and total electrostatic energy in the DSL for a Local Particle Pair Loop. Output: short-range potential energy* $u^{(sr)} = \sum_{i=1}^{N} U_i^{(sr)}$, $U_i^{(sr)} = q_i \phi_{r_c}^{(sr)}(\vec{r}_i)$ *and short-range forces* $\vec{F}_{r_c}^{(sr)}(\vec{r}_i)$.

```
double r0 = r.j[0] - r.i[0];
double r1 = r.j[1] - r.i[1];
double r2 = r.j[2] - r.i[2];
double r_sq = r0*r0 + r1*r1 + r2*r2; double r = sqrt(r_sq);
double mask = (r_sq < REAL_CUTOFF_SQ)? 1.0 : 0.0;
double r_m1 = 1.0/r;
double qiqj_rm1 = q.i[0] * q.j[0] * r_m1 * mask;
double term1 = qiqj_rm1*erfc(SQRT_ALPHA*r);
u[0] += 0.5*term1; // electrostatic energy
double term3 = -1.*r_m1*(qiqj_rm1 * TWO_SQRT_ALPHAOPI *
    exp(MALPHA*r_sq) + r_m1*r_m1*term1); // force
F.i[0] += term3 * r0; F.i[1] += term3 * r1; F.i[2] += term3 * r2;
```

Listing 5.2: *Python local* `ParticlePair` *loop creation and execution that reads* `ParticleDats` *for positions* $\vec{r}_i$ *and charges* q_i *and increments the* `ParticleDat` *for the force* $\vec{F}_{r_c}^{(sr)}$ *and* `GlobalArray` $u^{(sr)}$.

```
# Define kernel
kernel = Kernel('ewald_sr', kernel_code, kernel_consts)
# Define and execute pair loop
pair_loop = PairLoop(kernel=kernel, shell_cutoff=rc,
                     dat_dict={'r': Positions(access.READ),
                               'q': Charges(access.READ),
                               'F': Forces(access.INC),
                               'u': u_sr(access.INC)})
pair_loop.execute()
```

Long Range Potential

The evaluation of the long-range potential at position $\vec{r}_i$ can be written as

$$\phi^{(\mathrm{lr})}(\vec{r}_i) = \sum_{\vec{k}\neq 0}^{|\vec{k}|<k_c} C_{\vec{k}} A_{i,\vec{k}} \sum_{j=1}^{N} A^*_{j,\vec{k}} q_j, \tag{5.2}$$

$$= \sum_{\vec{k}\neq 0}^{|\vec{k}|<k_c} C_{\vec{k}} A_{i,\vec{k}} \hat{\rho}_{\vec{k}}, \tag{5.3}$$

$$\text{where} \quad \hat{\rho}_{\vec{k}} = \sum_{j=1}^{N} A^*_{j,\vec{k}} q_j, \tag{5.4}$$

$$A_{j,\vec{k}} = \exp\left(i\vec{k}\cdot\vec{r}_j\right), \tag{5.5}$$

$$C_{\vec{k}} = 4\pi/(V\vec{k}^2)\exp\left(-\vec{k}^2/(4\alpha)\right). \tag{5.6}$$

For an optimal value of α we showed in Section 4.2.1 that $k_c \propto N^{1/6}$. Hence the number of reciprocal lattice points N_k within a sphere of radius k_c is proportional to k_c^3. The expression in Equation (5.3) is essentially the product of a $N_k \times N$ matrix with a vector of length N followed by a multiplication by a $N \times N_k$ matrix.

Since the particles are distributed between p processors but all Fourier modes are computed and stored on each processor the computational cost is $\propto NN_k/p \propto N^{3/2}/p$. Every processor only calculates the contribution of all *locally* stored particles to every Fourier mode. Combining the contributions of all particles to each of the N_k Fourier modes therefore requires a global reduction of $N_k \propto N^{1/2}$ numbers, resulting in a total computational cost of $t = CN\left(\frac{N^{1/2}}{p} + rN^{-1/2}\log p\right)$ where the ratio $r \gg 1$ depends on the relative cost of computation and communication on a particular machine. We expect the code to scale well as long as $N \gg rp\log p$.

The computation of the long-range potential is split into two `ParticleLoop`s which correspond to the $N_k \times N$ and $N \times N_k$ matrix-vector products described above. The first iterates over all particles j and for each particle computes the contribution to $\hat{\rho}_{\vec{k}}$ defined in Equation (5.3) for all $|\vec{k}| < k_c$. An outline of the computational kernel is shown in Algorithm 22 (for brevity we do not show the corresponding C- and Python-code, but outline the access descriptors). We order the entries in the `GlobalArray` $\hat{\rho}_{\vec{k}}$ such that loops over reciprocal vectors $\vec{k}$ are vectorised by the compiler (as confirmed by the generated assembly code).

Algorithm 22: Computational kernel for the contribution to reciprocal space for a particle j.

Data: position $\vec{r}_j$ [READ], charge q_j [READ]
Result: Reciprocal space $\hat{\rho}_{\vec{k}}$ [INC]
foreach *reciprocal vectors* $\vec{k} \neq 0$ *such that* $|\vec{k}| < k_c$ **do**
  $\hat{\rho}_{\vec{k}} \mapsto \hat{\rho}_{\vec{k}} + A^*_{j,\vec{k}} q_j$
end

Note that the calculation of $\hat{\rho}_{\vec{k}}$ requires global reductions since each $\vec{k}$-component receives contributions from all particles in the system. This, however, is automatically handled by the code generation system and requires no explicit coding for the user who only writes the local kernel in line 2 of Algorithm 22. In our implementation we store copies of the entire vector $\hat{\rho}_{\vec{k}}$ on each MPI process and do not attempt a parallel domain decomposition in $\vec{k}$ space. Since the number of reciprocal vectors grows $\propto \sqrt{N}$ this does not lead to memory issues for moderately sized systems for which the Particle-Ewald method is competitive. Given the vector $\hat{\rho}_{\vec{k}}$, the electrostatic energies and forces are calculated as a second `ParticleLoop` using Equation (5.3) for each particle as in Algorithm 23.

Algorithm 23: Computational kernel to extract the long-range contribution from reciprocal space.

Data: Position $\vec{r}_j$ [READ], charge q_j [READ], $\hat{\rho}_{\vec{k}}$ [READ].
Result: Total electrostatic potential energy $u^{(\mathrm{lr})}$ [INC] and forces $\vec{F}_j^{(\mathrm{lr})} \equiv \vec{F}^{(\mathrm{lr})}(\vec{r}_j)$ [INC].
foreach *reciprocal vectors* $\vec{k} \neq 0$ *such that* $|\vec{k}| < k_c$ **do**
  $u^{(\mathrm{lr})} \mapsto u^{(\mathrm{lr})} + C_k A_{j,\vec{k}} q_j \hat{\rho}_{\vec{k}}$
  $\vec{F}_j^{(\mathrm{lr})} \mapsto \vec{F}_j^{(\mathrm{lr})} - i\vec{k} C_k A_{j,\vec{k}} q_j \hat{\rho}_{\vec{k}}$
end

The self-energy (not shown here) is calculated once at the beginning of the simulation and the cost of this operation is amortised over the total runtime.

Optimisations

We present results for a CPU only implementation where we exploit the regular structure of the reciprocal space. A point $\vec{k}$ in the reciprocal lattice is given by a linear combination of the reciprocal lattice vectors with integer coefficients:

$$\vec{k} = g_1\vec{G}_1 + g_2\vec{G}_2 + g_3\vec{G}_3, \quad \text{where} \quad g_1, g_2, g_3 \in \mathbb{Z}, \tag{5.7}$$

and $\vec{G}_1$, $\vec{G}_2$ and $\vec{G}_3$ are the reciprocal lattice vectors defined in Section 4.2. We assume the simulation cell is a cuboid and hence can rewrite contributions to reciprocal space as

$$A_{j,\vec{k}} = \exp\left(i\vec{k}\cdot\vec{r}_j\right), \tag{5.8}$$

$$= \exp\left(i\left[g_1\vec{G}_1 + g_2\vec{G}_2 + g_3\vec{G}_3\right]\cdot\vec{r}_j\right), \tag{5.9}$$

$$= \exp\left(ig_1\vec{G}_{1,x}\vec{r}_{j,x}\right)\exp\left(ig_2\vec{G}_{2,y}\vec{r}_{j,y}\right)\exp\left(ig_3\vec{G}_{3,z}\vec{r}_{j,z}\right), \tag{5.10}$$

$$= \exp\left(i\vec{G}_{1,x}\vec{r}_{j,x}\right)^{g_1}\exp\left(i\vec{G}_{2,y}\vec{r}_{j,y}\right)^{g_2}\exp\left(i\vec{G}_{3,z}\vec{r}_{j,z}\right)^{g_3}. \tag{5.11}$$

Hence we can compute entries of A by computing $\exp\left(i\vec{G}_{1,x}\vec{r}_{j,x}\right)$, $\exp\left(i\vec{G}_{2,y}\vec{r}_{j,y}\right)$ and $\exp\left(i\vec{G}_{3,z}\vec{r}_{j,z}\right)$ once and performing multiplication to compute successively higher powers. This optimisation is significant as computing exponentials is vastly more expensive than multiplication.

In a theoretical GPU implementation of Ewald summation the short-range component would be performed identically using a GPU *Particle Pair Loop*. For the long-range component, our algorithm to compute A on CPUs is not suitable for GPUs as for each charge j the contribution to $\rho(\vec{k})$ would cause write contention if we assigned each thread a charge. Alternatively, we could assign GPU threads to reciprocal space vectors but then we would be unable to efficiently apply the optimisation that avoids evaluating exponential functions.

5.2 Ewald Results

5.2.1 Computational Complexity

With a correct choice of α the Ewald method exhibits $\mathcal{O}(N^{3/2})$ computational cost. Figure 5-1 confirms this by plotting the time per iteration for a NaCl salt simulation against particle count N at a fixed density of 1 atom per $(2.5\text{Å})^3$. We include repulsive Lennard-Jones interactions to prevent the particle distribution from collapsing. However, for sizable particle counts the dominant computational cost is the electrostatic forces: for $N = 1.8{\cdot}10^5$ particles 87% of the time is spent computing Coulombic interactions. For all tests we set the error tolerance to 10^{-6} and vary the parameters α and r_c (which balance the work between the real- and Fourier-space) to minimise the runtime. For our framework the pair (α, r_c) takes values between $(0.062, 13.5\text{Å})$ for $N = 1728$ and $(0.013, 29.2\text{Å})$ for $N = 1.8 \cdot 10^5$. For DL_POLY_4 [41] we choose a cutoff value of $r_c = 10\text{Å}$. All runs are carried out on the "Balena" cluster; one node consists of two Intel E5-2650v2 8-core CPUs.

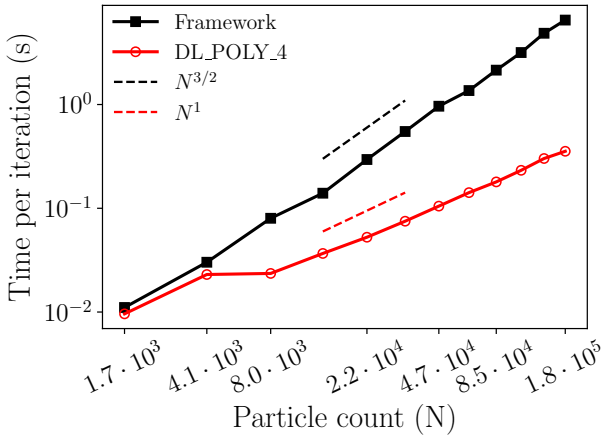

Figure 5-1: *Time per iteration against particle count for an NaCl system on a single 8 core CPU using OpenMP (our framework) or pure MPI (DL_POLY_4).*

Both implementations show at least the expected scaling with a power of N. For small particle numbers the SPME method used by DL_POLY_4 is in the same ballpark as our implementation. The SPME method obviously outperforms our method for larger particle counts where it is an order of magnitude faster.

5.2.2 Strong Scaling

To study the parallel scalability we set the number of particles to $N = 3.3 \cdot 10^4$ in a box of size 80Å $\times$ 80Å $\times$ 80Å (at the same density as in Section 5.2.1) and increase the core count. The spatial domain cannot be decomposed into regions of side length less than the cutoff r_c which prevents repeating the runs in Section 5.2.1 on more than one node. To address this, we fixed $r_c = 19$Å ($r_c = 10$Å for DL_POLY_4) at the price of using a non-optimal value of α (0.032 instead of 0.023). This allows our MPI-only implementation and DL_POLY_4 to scale to 64 cores and we find that it has no negative impact on the runtime on one CPU. To scale beyond this limit we use a hybrid MPI+OpenMP scheme with one MPI process per CPU socket to run on up to 256 cores. To quantify any potential performance loss due to the non-optimal value of α, we also include the relevant data point with $(\alpha, r_c) = (0.023, 22.1\text{Å})$ from Figure 5-1.

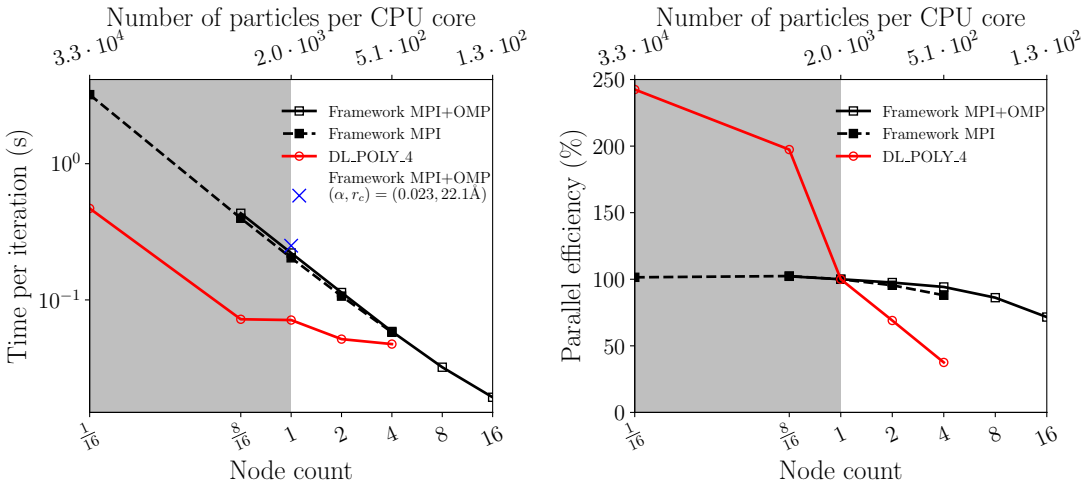

Figure 5-2: *Strong scaling experiment of an NaCl system comparing our implementation, labeled as "Framework", with DL_POLY_4. Time per iteration (left) and parallel efficiency relative to one 16-core node (right). Time taken is recorded for* $3.3 \cdot 10^4$ *charges over 300 Velocity Verlet iterations. Short-range Lennard-Jones interactions are enabled with a cutoff of* $3\mathring{A}$.

Both the MPI and MPI+OpenMP implementations exhibit decent scaling to 16 nodes (256 cores). DL_POLY_4 is faster overall on smaller core counts but does not scale to larger core counts. The MPI+OpenMP execution of Algorithm 23 on one node achieved an average of 34% of peak floating point vector performance, our implementation of this algorithm exhibits an arithmetic intensity of approximately 2. The computationally most expensive component is the loop over all Fourier modes $\vec{k} = (k_1, k_2, k_3)$. This has been vectorised over the four quadrants with $(\text{sign}(k_1), \text{sign}(k_2)) = (+,+)$, $(+,-)$, $(-,+)$ and $(-,-)$ and we confirmed that the Intel compiler indeed generates packed vector instructions.

5.3 Fast Multipole Method

The computational work of the FMM is split into two components; the direct particle-particle interactions and the indirect interactions through multipole and local expansions. The direct interactions are extremely similar to the inter-particle potentials discussed in Section 1.1.3 with an additional constraint on which particle pairs are considered. The indirect interactions require data structures to represent the hierarchical mesh and expansion coefficients stored in cells on each level of the tree. Furthermore, we require efficient implementations of the functions that create and manipulate multipole and local expansion coefficients. Our implementation approach is to use the flexibility of Python to pre-compute constant objects, such as rotation matrices, and implement C code for the computationally expensive components.

The implementation follows Algorithms 20 and 21 and requires the following functionality:

1. Determine particle cells and contributions to multipole expansions.

2. Implementation of multipole to multipole translation operation.
3. Movement of multipole expansions from a level to the next coarsest level.
4. Movement of local expansions from a level to the next finest level.
5. Implementation of local to local translation operation.
6. Implementation of multipole to local translation operation.
7. Evaluation of a local expansion at the position of each charge.
8. Direct charge to charge interactions.

5.3.1 Indirect Interactions

Octal Tree

We use the term "Octal Tree" (OT) to refer to the hierarchy of mesh levels imposed on the simulation domain. We implement data structures to describe Octal Trees such that data can (1) be attached to cells, (2) moved between levels in the tree and (3) communicated between adjacent cells on the same level. An OT is distributed across MPI ranks in a similar manner to the domain decomposition approach described in Section 3.1.2 in that on the finest level of the OT each MPI rank owns the set of cells that approximately matches the owned sub-domain of the simulation domain. This matching is approximate as the mesh cells are discrete objects whereas the sub-domains of the simulation domain are created by simply assigning each MPI rank an equal portion of the domain.

As each cell in a level of the OT is assigned a unique owner not all MPI ranks will own cells on the coarse levels. For example, at the top of the OT, on the coarsest level, there is only a single cell and hence only one MPI rank will own a cell. Secondly, we group cells such that the child cells of any given cell are all owned by the same MPI rank, the owning MPI rank of the child cells can be different to the owning MPI rank of the parent. This grouping reduces the amount of required MPI communication as data transfer between child cells and parent cells involves at most two MPI ranks, secondly, this grouping reduces the complexity of the code. An example 2D decomposition is given in Figure 5-3 where an OT with 3 mesh levels is distributed over 4 MPI ranks.

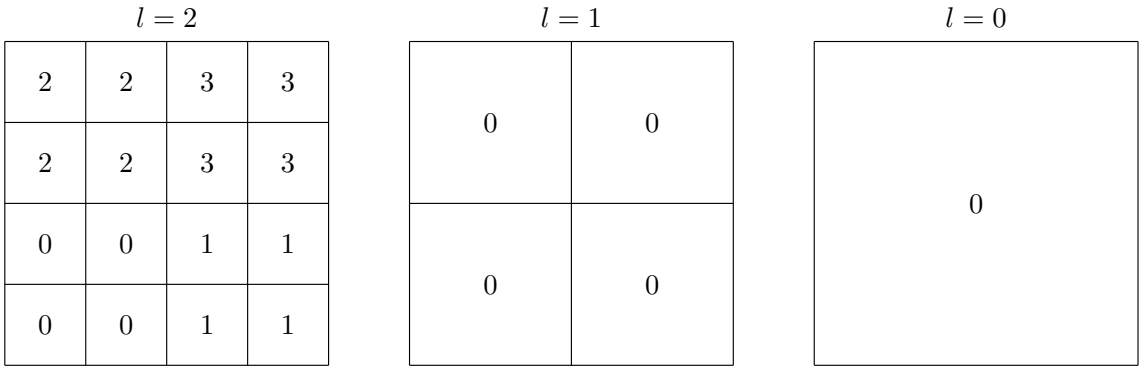

Figure 5-3: *2D OT distributed over 4 MPI ranks, MPI ranks are labeled in the centre of the cells. On level $l = 1$ all cells are owned by rank 0 to keep all the children of cell 0 on level 0 on the same MPI rank.*

The Python `OctalTree` class stores an OT as a collection of mesh levels, each mesh level is decomposed over the maximum possible number of MPI ranks using the policy described above. A mesh level is stored as an instance of the `OctalGridLevel` class which records exactly how the mesh level is distributed over MPI ranks by providing the local size and offset on each MPI rank. Furthermore, the `OctalGridLevel` provides the map from global cell index to local cell index and global cell index to owning MPI rank. The multipole to local translation operation will require the communication of data stored in cells between MPI ranks on a mesh level, each `OctalGridLevel` provides an MPI Cartesian Communicator from the set of MPI ranks that own cells on the level.

Data Storage On An Octal Tree

The standard FMM algorithm requires storage for a p-term multipole or local expansion in each cell on each mesh level. We provide the `OctalDataTree` class to store data in each cell on each level of the octal tree. This storage is realised as a 4D numpy array where the first 3 dimensions are determined by the size of the owned block of cells on each rank and the last dimension is the number of elements in each cell as chosen by the user. This class has three variants to facilitate the implementation of the FMM algorithm:

"plain" If the MPI rank owns a (C_x, C_y, C_z) block of cells and requests storage for N_e elements then the allocated numpy array has dimensions (C_x, C_y, C_z, N_e). We use this type to store local expansions.

"parent" If the MPI rank owns a (C_x, C_y, C_z) block of cells and requests storage for N_e elements then the allocated numpy array has dimensions $(C_x/2, C_y/2, C_z/2, N_e)$. This data structure provides storage for the temporary arrays required in the Multipole to Multipole and Local to Local translations. For example, in a Multipole to Multipole translation the operator $\mathcal{T}_{\mathrm{MM}}$ is applied to each of the 8 child cells of a parent and the result constructed in the temporary array. The temporary array is then communicated to the MPI rank which owns the parent cell. Hence this type acts as a staging area for communication between levels of an OT.

"halo" If the MPI rank owns a (C_x, C_y, C_z) block of cells and requests storage for N_e elements then the allocated numpy array has dimensions $(C_x+4, C_y+4, C_z+4, N_e)$. For a cell i the multipole to local operation requires data stored in the cells which are the children of cells adjacent to the parent of i as in Figure 4-9. However, as we distribute the cells on a level across MPI ranks the cells in the interaction list of a given cell may be stored on a different MPI rank. Hence the "plain" type is padded by four cells in each dimension to provide the required storage for MPI communication within a mesh level. `OctalDataTrees` of this type provide a `halo_exchange` method which automatically communicates the required data on a requested mesh level.

Inter-level communication occurs in both directions, multipole expansion coefficients are combined and traverse from fine levels to coarse levels. On each level $l > 1$ we apply the following process:

1. On mesh level l compute and store multipole expansions $\Phi_{l,i}$ in a "halo" `OctalDataTree` called `data_halo`.

2. Perform the halo exchange on `data_halo` to communicate the multipole expansions within the level l ready for the multipole to local stage of the downward pass.

3. Apply the $\mathcal{T}_{\mathrm{MM}}$ operator with input expansions in `data_halo` and store output expansions in a `OctalDataTree` of type "parent" on level l called `data_parent`.

4. Perform inter-level communication by copying the expansions stored in `data_parent` on level l into `data_halo` on level $l-1$.

Local expansion coefficients move in the coarse to fine direction in the downward pass. On each level $l > 1$ we apply the following process:

1. On mesh level l perform inter-level communication that copies the local expansions $\Psi_{l-1,i} = \bar{\Psi}_{l,i}$ from the previous level. The source local expansions are stored in a `OctalDataTree` of type "plain" called `data_plain` and are copied into `data_parent`.

2. Apply the $\mathcal{T}_{\mathrm{LL}}$ operator with source expansions $\bar{\Psi}_{l,i}$ in `data_parent` on level l and output expansions in `data_plain` on level l.

3. Apply the $\mathcal{T}_{\mathrm{ML}}$ operator on level l, source multipole expansions are stored in `data_halo` from the upward pass and output local expansions $\Psi_{l,i}$ are accumulated in `data_plain`.

Translation Operations

All three translation operations that manipulate expansions are implemented as C libraries. We implement shared memory parallelism with OpenMP in all FMM C libraries to increase the parallel efficiency of the implementation. Parallel efficiency is increased as more CPU cores can be allocated per mesh level, in particular on the coarser mesh levels, for example, mesh level $l = 1$ which is owned by a single MPI rank and contains eight

cells. Without a form of shared memory parallelism all computation with cells on these coarse levels would be computed by a single CPU core. The translation operations are implemented by applying the following equations:

$\mathcal{T}_{\text{MM}}$ Equation (4.133)

$\mathcal{T}_{\text{LL}}$ Equation (4.141)

$\mathcal{T}_{\text{ML}}$ Equation (4.160)

We exploit the structure of the mesh hierarchy to reduce the computational work, in particular the fact that all cells have identical shape and adjacent cell structure. In the $\mathcal{T}_{\text{MM}}$ and $\mathcal{T}_{\text{LL}}$ operations the values of $Y_n^m(\alpha, \beta)$ are required where α and β are the angular component of the translation vectors. Similarly, we require the matrices that apply the rotation operators $\mathcal{R}_z(\beta)$ and $\mathcal{R}_y(\alpha)$, the first of which we apply in a matrix free manner as it is diagonal and elements can be cheaply constructed.

If $\mathcal{W} = \left\{\vec{x} : \vec{x} \in \{-3, \ldots, 3\}^3, |\vec{x}|_\infty > 1\right\}$ denotes the set of all possible translation vectors and $(r_{\vec{w}}, \alpha_{\vec{w}}, \beta_{\vec{w}})$ is the spherical coordinate representation of an offset vector $\vec{w}$, then we pre-compute and store the values of $Y_n^m(\alpha_{\vec{w}}, \beta_{\vec{w}})$ and $d_n^{m,m'}(\alpha_{\vec{w}})$ for all $\vec{w} \in \mathcal{W}$. These pre-computed values are valid for any cubic domain as they are independent of cube size and in principle could be cached on persistent storage. The final matrix we pre-compute is R_n^m which, as discussed in Section 4.3.2, is applied in the multipole to local translation of all well separated periodic images.

5.3.2 Direct Interactions

For each particle i we compute and store the containing cell c_i when the contribution to the multipole expansion Φ_n, c_i is computed, the value c_i is stored as a particle property in a `ParticleDat`. By storing these values in a `ParticleDat` the halo exchange machinery developed for the *Local Particle Pair Loop* implementations in Chapter 3 is simply executed to exchange the particle positions $\vec{r}_i$, particle charges q_i and particle cells c_i between MPI ranks.

For a charge i the existing Local Particle Pair Loop implementations were designed for efficiently constructing the list of pairs (i, j) where $|\vec{r}_i - \vec{r}_j| < r_c$, i.e. all neighbouring charges j within a sphere of radius r_c. For the direct interactions, the volume containing the relevant neighbours of i is not a sphere centred on i but the cube formed as the union of the cube containing i and the 26 adjacent cubes.

We use a C library specifically written for the direct interactions to compute the inter-charge interactions. The library uses the cell list method described in Chapter 3 to construct a map from cells to contained charges. The library then applies a cell by cell approach to compute the direct interactions as in Algorithm 24. This approach attempts to minimise the reading and storing of particle data and arranges the temporary values in an non-interlaced manner which is more efficient for the vector floating point units in modern CPUs.

Algorithm 24: Cell by cell method to compute direct charge to charge interactions.

Data: Charge positions $\vec{r}_i$, charges $\vec{q}_i$ and cells c_i. Cell to charge map $\mathcal{C}$. Set of cells $\mathcal{D}$ which are owned by this MPI rank.

Result: Direct potential energy $\mathcal{U}$ and charge forces $\vec{f}$.

$\mathcal{U} \leftarrow 0$

for *cell* $d \in \mathcal{D}$ **do**

  Populate vector of positions $\vec{r}_d$ and charges q_d containing all $i \in \mathcal{C}(d)$

  Initialise vector of forces $\vec{f}_d \leftarrow \vec{0}$

  for *cell* d' *adjacent to* d **do**

    Populate vector of positions $\vec{r}_{d'}$ and charges $q_{d'}$ containing all $i \in \mathcal{C}(d')$

    for $i \in \mathcal{C}(d)$ **do**

      for $j \in \mathcal{C}(d')$ **do**

        $\mathcal{U} \leftarrow \mathcal{U} + \frac{q_d^i q_{d'}^j}{2|\vec{r}_d^i - \vec{r}_{d'}^j|}$

        $\vec{f}_d^i \leftarrow \vec{f}_d^i + \left(\vec{r}_d^i - \vec{r}_d^j\right) \frac{q_d^i q_{d'}^j}{|\vec{r}_d^i - \vec{r}_{d'}^j|^3}$

      end

    end

  end

  for $i \in \mathcal{C}(d)$ **do**

    for $j \in \mathcal{C}(d), j \neq i$ **do**

      $\mathcal{U} \leftarrow \mathcal{U} + \frac{q_d^i q_{d'}^j}{2|\vec{r}_d^i - \vec{r}_{d'}^j|}$

      $\vec{f}_d^i \leftarrow \vec{f}_d^i + \left(\vec{r}_d^i - \vec{r}_d^j\right) \frac{q_d^i q_{d'}^j}{|\vec{r}_d^i - \vec{r}_{d'}^j|^3}$

    end

  end

  Write new forces to `ParticleDat`:

  for $i \in \mathcal{C}(d)$ **do**

    $\vec{f}_i \leftarrow \vec{f}_i + \vec{f}_d^i$

  end

end

5.4 Fast Multipole Method Results

Configuration And Parameter Selection

We compare the performance of our FMM implementation with the FFT accelerated Ewald approach in DL_POLY_4. Our test configuration is based on the two ion NaCl simulation TEST01 [14] from the DL_POLY test suite. The Sodium ions (Na) carry a charge of +1, conversely, the Chloride ions carry a charge of -1. In our configuration the ions interact with a short range Lennard-Jones potential that prevents oppositely charged ions from collapsing onto each other. We set the short range cutoff at 4Å, which is small

enough to have negligible impact on the time per iteration. Unlike the FMM method, the computational cost of FFT based Ewald approaches is dependent on the volume of the simulation domain, to provide a fair comparison we duplicate the density of the original configuration. The original configuration places ions in a simple cubic lattice of alternating species with a lattice constant of 3.3Å in each dimension.

To configure the accuracy of the DL_POLY Ewald implementation the user specifies a desired precision which may or may not be achieved in practice. To configure the accuracy of the FMM implementation the number of expansion terms are chosen. To fairly compare the performance of the two different implementations we choose input precision parameters that provide similar measured output accuracy for the potential energy of the system.

Output accuracy from the FMM implementation and DL_POLY is estimated by creating 10 pseudo random, dipole free and net charge free configurations of 10^6 ions. To create a dipole and net charge free configuration we first created a cubic configuration of 125000 ions in a 50x50x50 cubic lattice with a 3.3Å lattice spacing. The position of each ion is then perturbed in each coordinate direction by a sample from a uniform random distribution with minimum $-3.3/2$ and maximum $3.3/2$, this adds disorder to the system whilst ensuring no charges overlap. The 50x50x50 base configuration is duplicated and reflected in the appropriate planes to create a 100x100x100 configuration that is dipole free. The "true" system potential energy for each configuration is computed with our Classical Ewald implementation presented in Sections 4.2 and 5.2. For each configuration, implementation and input parameter we take the maximum error over the 10 configurations as the estimated error.

The estimated output errors from both implementations is plotted in Figure 5-4. We (as fairly as possible) equate output errors between our FMM implementation and DL_POLY by choosing the output error that corresponds to a DL_POLY input precision of 10^{-6}, which is a typical precision chosen by end users. Based on these results we use 10 expansion terms (spherical harmonics of order 0 to 9) in all multipole and local expansions.

The number of FMM levels $\mathcal{L}$ is an integer quantity chosen as $\mathcal{L} = \lfloor \log_8(\alpha N) \rfloor$, where α is a parameter that tunes the number of levels to balance the work between direct charge-charge interactions and indirect interactions through expansions. The parameter α is dependent on the HPC hardware and the number of expansion terms used, using more expansion terms increases the cost of the indirect interactions. For 10 expansion terms we determined that $\alpha = 0.2$ was a near optimal value to use. Note that the computational cost is dependent on the number of levels in the tree, but the output accuracy is independent of the number of levels.

Both DL_POLY and our FMM implementation were compiled with the Intel compiler version 18.0.0 20170811 and Open MPI version 3.0.0. The FMM implementation was launched in two modes; MPI only and hybrid MPI+OpenMP using 1 MPI rank and 8 OpenMP threads per socket. DL_POLY is a MPI only program.

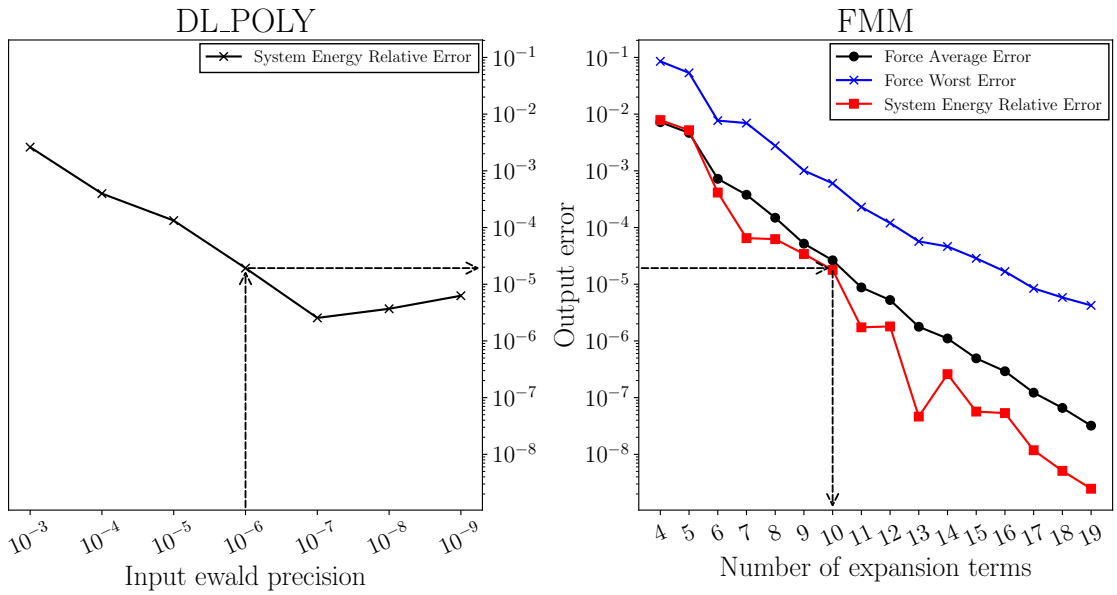

Figure 5-4: *(Left) relative error in system potential energy from DL_POLY against input precision, dashed black arrows indicate the output error for an input precision of* 10^{-6}*. (Right) relative error in system potential energy and absolute error in particle forces against number of expansion terms used for all expansions, dashed arrows indicate the number of expansion terms required to meet the DL_POLY output error in the left plot. Average force error is computed as the average error over all charges over all component directions. Worst force error is computed as the maximum error over all ions and all component directions.*

Strong Scaling

To test the strong scaling performance of the FMM implementation we perform 200 Velocity Verlet integration steps of a system containing 10^6 charged particles and a system of $160^3 \approx 4{\cdot}10^6$ charged particles. The parameters of the simulation are identical to the NaCl configuration used to estimate the error in the potential energy. The initial configuration is a cubic lattice of alternating particle species with a lattice spacing of 3.3Å.

We used 5 mesh levels for the simulation containing $1 \cdot 10^6$ charges which results in 8^4 cells on the finest mesh level. As our implementation groups cells such that all child cells of a given cell are on the same MPI rank, this results in $8^3 = 512$ groups of cells on the finest level to be distributed across MPI ranks, hence with plain MPI execution the implementation has a hard strong scaling limit at 512 CPU cores. The simulation containing $4{\cdot}10^6$ charges uses 6 mesh levels and hence in MPI only execution mode exhibits a hard strong scaling limit at 4096 cores, more cores than our HPC facility contains.

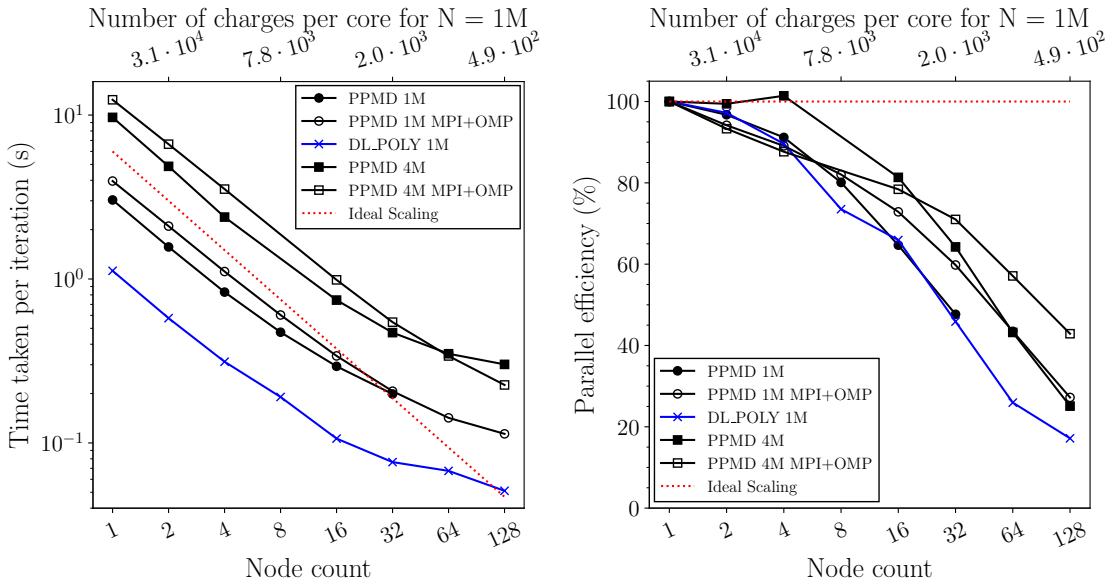

Figure 5-5: *Strong scaling comparison between our FMM implementation, labeled as "PPMD", and DL_POLY FFT based Ewald. (Right) Time taken per Velocity Verlet iteration. (Left) Parallel efficiency as defined in Equation (3.9) computed relative to 1 node. One node consists of two Intel Xeon E5-2650v2 CPUs (16 cores per node). Time per iteration and parallel efficiency is recorded for a system containing* 10^6 *charges and a system containing* $4 \cdot 10^6$ *charges.*

The 10^6 particle run indicates that DL_POLY is ≈ 3 times quicker than our FMM implementation for this particular simulation. Considering the maturity of DL_POLY we regard the performance of our FMM implementation to be reasonably good for an initial implementation. The parallel efficiency results indicate that our FMM implementation does not exhibit unreasonable performance degradation when compared to an existing code. Furthermore, these results indicate that applying a hybrid MPI+OpenMP model to our FMM implementation does increase parallel efficiency in the strong scaling limit. When not in the strong scaling limit, the efficiency of the hybrid approach is approximately a third less than a pure MPI approach. The hybrid approach introduces atomic operations not found in the pure MPI implementation, these lead to reduced intra-node parallel efficiency as described by Amdahl's Law [6].

Weak Scaling

We set up a weak scaling experiment where the number of charges in the simulation per CPU core remains fixed. For N particles the FMM method exhibits a computational complexity that is asymptotically $\mathcal{O}(N)$, hence if the number of charges is increased at the same rate as the number of CPU cores then the time per FMM evaluation should remain constant. The time taken per iteration is expected to vary as the number of mesh levels is an integer quantity which is fixed for an interval of charge numbers N. We record the time taken per Velocity Verlet iteration for particle counts in the range $1 \cdot 10^6$ to $128 \cdot 10^6$ for pure MPI execution and hybrid MPI+OpenMP execution. On 64 nodes we exhausted available memory for MPI only execution due to memory inefficiencies in non-

FMM related portions of code, for node counts larger than 32 we investigated only hybrid execution.

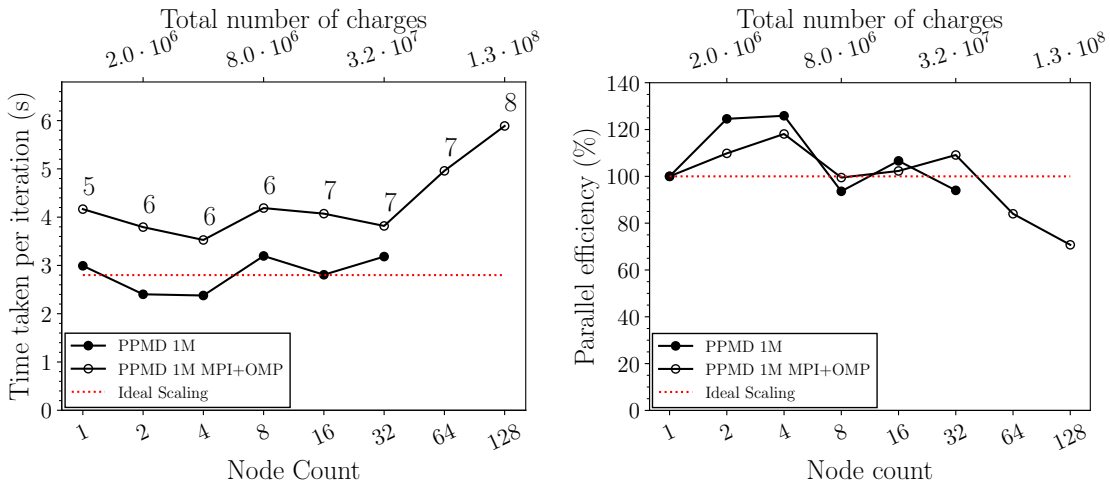

Figure 5-6: *Weak scaling test of our FMM implementation. (Right) Time taken per Velocity Verlet iteration, floating numbers indicate the number of levels in the octal tree. (Left) Parallel efficiency as defined in Equation (3.10) computed relative to 1 node. One node consists of two Intel Xeon E5-2650v2 CPUs (16 cores per node).*

The weak scaling results indicate that the FMM implementation performs approximately linearly over the particle counts and CPU core counts we investigated. The results indicate a slight upward trend that we expect is related to how our octal tree distributes computational work over MPI ranks. Each level in the tree is decomposed over as many MPI ranks as possible, on the finest level all MPI ranks own cells and perform useful work. Furthermore, on the coarser levels the maximum possible number of MPI ranks is relatively low, level 0 is owned by one MPI rank. In this implementation each level in the downward pass of the FMM algorithm is performed sequentially, for example, the MPI ranks that own the cells on mesh level 2 must wait until the 1 MPI rank that owns level 1 completes the downward pass on level 1. Hence we expect a degree of parallel inefficiency in the weak scaling experiment purely from our MPI decomposition approach. Future work could investigate a different method to assign cells to MPI ranks to perform the downward pass.

CHAPTER 6

CONCLUSION AND FUTURE WORK

6.1 Summary Of Work

The focus of this thesis is the performance-portable implementation of MD algorithms for HPC facilities. We address the issue that the efficient implementation of MD algorithms requires expertise from the domains of physics and chemistry and from the computational science domain.

In our literature review and preliminary research we observed that existing MD libraries typically were designed and implemented in a monolithic manner over many years. These libraries have a large number of features and are typically efficient on the hardware they were designed for. However, the monolithic nature of these libraries creates a portability challenge when new hardware architectures emerge as often algorithms that are efficient on the first architecture are not efficient on the second. Furthermore, re-writing large code bases for emerging architectures is an expensive process that requires detailed knowledge of both the scientific domain and the target hardware architecture.

We identified that alongside simulations, domain specialists perform analysis techniques to produce quantitative outputs from simulations tailored to their area of interest. Implementing new analysis techniques within existing MD software typically requires detailed knowledge of the inner workings of software packages, frequently with large code bases, and hence users often dump simulation outputs for post processing. The post processing of simulation output may well be more computationally expensive than the simulation itself and implementing analysis in a parallel manner may well be outside the skill set of domain specialists.

To address those issues, we applied a separation of concerns based approach shown to be successful in other areas of science. This approach separates the concerns of the domain specialists from those of computational scientists. The separation is enabled by our abstraction that allows a high level description of many algorithms and methods involving particles. Our abstraction is conceptually simple yet is highly flexible for describing par-

ticle data and global data alongside looping mechanisms to execute one- and two-particle kernels.

We created a Python-embedded DSL that implements our abstraction and serves as an input to our code generation system. The DSL requires users to implement desired operations within a subset of the C language. Using C as the DSL language allows low-level control within the kernel and enables a code generation system to be constructed based on templating. However, by using the C language we do not fully separate domain specialists from low-level considerations. For example, a user can write a non-compliant kernel within the DSL that does not compile, alternatively, a user could attempt to access memory out of bounds. We discuss future work to address these issues in Section 6.2. While the user does have to express some operations in C, our implementation successfully abstracts parallel looping operations away from the user.

We demonstrated that within the abstraction existing methods can be implemented and new methods can be described. Our DSL allows simulations to be constructed within a high-level language that has a large repository of other scientific packages that users can interface with. We demonstrated the capabilities of the abstraction by reimplementing functionality found in existing MD packages and by implementing non-trivial analysis algorithms.

The abstraction serves as input to our code generation framework that produces performant C code for two currently dominant HPC architectures. We demonstrated that a single input Python script can be used to target these two architectures efficiently with minimal changes. Hence our abstraction and DSL combination is portable between these two architectures. This portability between architectures is important as it enables algorithms to be described once and reused in a hardware independent manner. Furthermore, we are confident that due to the flexibility of our approach there is a reasonable chance that future hardware architectures could be targeted by a code generation system with our abstraction as input.

We compared our code generation approach with existing libraries for non-bonded interactions. The strong scaling results presented in Section 3.3 demonstrate that our generated code and its surrounding implementation is highly competitive with existing libraries on 1024 CPU cores and multiple GPUs. The corresponding weak scaling results demonstrate that our implementation scales well to systems containing 5.2×10^8 particles on 1024 CPU cores and 8.2×10^6 particles on 16 GPUs.

Our structure analysis results demonstrate that the abstraction is sufficient to implement algorithms that are more complex than inter-particle potentials. We show that our framework enables parallel on-the-fly analysis to be described and performed within a simulation. Although we demonstrated this functionality within a simulation, a user could implement a Python script that only performs analysis and conduct post-processing of data from a simulation.

Electrostatic interactions are computationally expensive and prevalent in MD simulations. We provide parallel implementations of two existing algorithms that compute long-

range interactions. The first method we investigated is the Ewald summation method, which has a non-optimal complexity but enabled our framework to compute electrostatic interactions. Our Ewald method is entirely implemented within our abstraction and DSL combination and scales extremely well in a strong scaling scenario. In Section 5.1 we demonstrated that our Ewald implementation is capable of scaling a small system (3.3×10^4 particles) across 512 CPU cores with a high parallel efficiency.

We investigated the FMM as the computational complexity is linear in the number of charges, and this implementation extended the long-range capabilities of our framework. The improved complexity in comparison to Ewald summation enables our framework to compute electrostatic interactions between charges in significant sized systems. Our strong scaling results in Section 5.4 demonstrate that our FMM implementation has comparable performance with an existing code that implements an FFT accelerated Ewald method on 2048 CPU cores. Furthermore, our weak scaling results demonstrate that our implementation has a near linear computational complexity.

6.2 Critical Assessment And Future Work

We described an abstraction for one- and two-particle kernels but did not investigate the case of general n-body potentials. In principle, the current abstraction can describe n-body potentials as shown by our implementation of the CNA in Section 2.3.2, but the method we use in this example may well not be the most efficient method in practice. In future the abstraction and DSL should be extended to allow n-body kernels to be described, and we do not think this is a greatly challenging extension to create. Furthermore, the bonded interactions between a group of particles that form a molecule are of interest to computational chemists and these interactions are described by rigid-body dynamics. There exist algorithms to compute these rigid body dynamics yet we have not discussed them in this thesis, as in the case of n-body kernels, we envisage that these dynamics could be described as an extension to the abstraction.

Furthermore, simulations often contain multiple species of particles, in our existing framework users are expected to distinguish between particle species by attaching labels. Distinguishing between particle types by using particle data is a sufficient way to implement multiple species simulations, however, we expect that a more efficient approach is to modify our framework such that multiple species interactions are explicitly handled. By explicitly handling multiple species at the DSL level the framework may make high-level reasoning that leads to further optimisations which are not currently not possible.

For example, consider a simulation containing two particle types A and B and suppose that particles of type A have a fixed position and particles of type B are free to move. As particle of type B move the current framework will perform halo-exchanges for all particles including those of type A, furthermore the A-A interactions are constant and could be computed once. As implementing this functionality could lead to significant efficiency improvements for many simulations we consider multiple species support to be a prime

candidate for future work.

The nature of the separation of concerns approach means that future work can improve the code generation framework without negatively impacting domain specialists. We provided implementations of neighbour list and cell by cell approaches for the *Local Particle Pair Loop* and should investigate further cell by cell approaches where cell sizes may be smaller than the interaction cutoff. These methods based on smaller cell sizes have the potential to combine the advantages of neighbour lists with the advantages of cell by cell methods.

Currently the algorithm and hardware used to execute a *Local Particle Pair Loop* is chosen by the user yet for a general kernel the most optimal algorithm and hardware type is non-obvious. In future the framework could exploit the fact that in a simulation most of the particle pair-loops are performed many times within a time stepping loop. To select the optimal algorithm the framework may simply trial different algorithms within the first few iterations. If a user is performing a multiple hour or day simulation it is worthwhile spending compute time within the first few iterations to select an optimal algorithm. There is scope for future implementations of the framework to use multiple hardware architectures in a heterogeneous manner and furthermore automatically determine from a set of available architectures which architecture is most efficient for a given kernel and looping type.

Currently the kernels in the DSL are written by the user in a subset of C and an experienced user can write highly efficient kernels. However, C is a low-level language that is technical to write and exhibits unforgiving behaviour from incorrect input. Hence by using C as the kernel input language we do not completely separate users from low level considerations. The Unified Form Language (UFL) [5] used by the FEniCS [4] and Firedrake projects completely separates users from low-level concerns by allowing users to describe algorithms in a symbolic-like form. The symbolic representation of operations is translated into low-level code automatically, in the case of the Firedrake project, by using code generation.

Future work should develop a symbolic-like language, inspired by UFL, such that users are able to write a symbolic description of a kernel that is automatically translated into a optimised C-kernel. Implementing a working version of this functionality is very achievable with existing Python libraries, however, implementing a framework that generates highly optimal C code is an active area of research. The development of this symbolic kernel language, although non-trivial, would greatly reduce the barrier-to-entry of our framework for domain specialists. In general, lowering the barrier-to-entry is a crucial process to increase the adoption rate of a software project, especially crowed markets.

We investigated CPU and GPU architectures as they are prevalent and readily available. Very recent CPU models combine a traditional CPU with an FPGA device in the same socket, if these devices become user programmable and provided they provide sufficient performance then there is significant scope to extend the capabilities of the framework to FPGA devices. Current FPGA devices are rare within HPC facilities and do not have

a large enough user base to consider supporting.

With the initial availability of the Isambard [33] HPC facility we investigated using our existing code generation framework on the Intel Xeon Phi architecture and Cavium ThunderX2 ARM processors [13]. Due to highly sub-optimal results, uncertain future support of and small user base we do not expect to investigate the Xeon Phi platform in the future. Conversely, our initial experimentation on the ThunderX2 platform suggests that these CPUs could be highly competitive with the Intel CPUs that dominate the market and that future work should investigate generating efficient code for these cores.

Our current code generation system produces efficient code in comparison to existing codes but could benefit from further optimisations. In particular we make no attempt to perform kernel fusion which would combine user written kernels that are compatible to decrease increase the overall computational work of a simulation. Further improvements to parallel efficiency could be realised by performing halo-exchanges in parallel with computation, a particle pair-loop could be executed over the interior of a sub-domain whilst halo exchanges occur to communicate boundary data. This optimisation is likely to be impactful on accelerator devices such as GPUs where the latency of communication is higher.

In our FMM results section (Section 5.4) we conclude our FMM implementation is competitive, however, we identify that the parallel performance could be improved with a different distribution of work across MPI ranks and this could form the basis of future work. After the upward pass has been performed, all multipole to local translations on all levels can be performed simultaneously. The local to local translations would then be performed sequentially and combined with the results of the multipole to local translations. This reassignment of cells to MPI ranks reduces the number of idle MPI ranks in the downward pass by providing a more efficient distribution across MPI ranks of all cells in the octal tree. Furthermore, future work could investigate a GPU implementation of the FMM algorithm that performs both the direct and indirect interactions on the GPU.

We focused on the MD approach to compute ensemble averages of quantities, MC is a popular technique that could be incorporated into the abstraction with the addition of MC specific properties and looping types. For example, an extension to the abstraction could allow proposed changes to the properties of a single particle and provide looping operations that assumed the properties of all other particles remained constant. For our MD focused implementation we apply a domain decomposition approach for parallelisation across MPI ranks, this approach is unlikely to be optimal for MC.

In the immediate future we shall investigate using the FMM to compute electrostatic interactions in a particular type of MC known as kinetic Monte Carlo (KMC). In this approach the simulated system contains charged particles that occupy sites in the domain. An iteration of the algorithm proposes moving each charge one-by-one to all of the empty sites in the immediate vicinity of its current site and for each of these proposed sites the potential energy of the whole system is computed. One of the proposed moves is then randomly selected based on the change of system potential energy. As the FMM is a com-

putationally optimal and highly flexible method for computing electrostatic interactions, we are investigating how the method can be adapted to efficiently compute the change in energy due to a single charge moving. Our initial implementation of an adapted FMM suggests that our approach offers optimal computational complexity for proposing and accepting moves.

APPENDIX A

APPENDICES

A.1 Largest Subcluster Algorithm

Algorithm 25 can be used to calculate the size of the largest connected component of a graph given by a set of edges $\mathcal{E}$. For this the edges in each subgraph are counted with a breadth-first like traversal, counting and removing all visited edges in the process.

Algorithm 25: Calculate maximal cluster size.

Data: Graph defined by a set of edges $\mathcal{E}$.
Result: $S_{\max}$, the size of the largest cluster.
$S_{\max} \leftarrow 0$
while $\mathcal{E} \neq \emptyset$ **do**
  $S \leftarrow 0$
  Pick some edge $(v_1, v_2) \in \mathcal{E}$
  $\mathcal{Q} \leftarrow \{v_1\}$
  while $\mathcal{Q} \neq \emptyset$ **do**
    Pick some $v \in \mathcal{Q}$ and remove it from $\mathcal{Q}$
    $\mathcal{P} \leftarrow \{(v, w) \in \mathcal{E}\}$
    $\mathcal{Q} \leftarrow \mathcal{Q} \cup \{w : (v, w) \in \mathcal{P}\}$
    $S \leftarrow S + |\mathcal{P}|$
    Remove all edges $e \in \mathcal{P}$ from $\mathcal{E}$
  end
  $S_{\max} \leftarrow \max\{S, S_{\max}\}$
end

A.2 Negative Binomial Expansion

If $|z_0/z| < 1$ then

$$(z - z_0)^{-k} = \sum_{l=0}^{\infty} \binom{k+l-1}{l} z_0^l z^{-k-l} \tag{A.1}$$

$$= \sum_{l=k}^{\infty} \binom{l-1}{l-k} z_0^{l-k} z^{-l} \tag{A.2}$$

$$= \sum_{l=k}^{\infty} \binom{l-1}{k-1} z_0^{l-k} z^{-l} \tag{A.3}$$

as

$$\binom{l-1}{l-k} = \frac{(l-1)!}{[(l-1)-(l-k)]!(l-k)!} \tag{A.4}$$

$$= \binom{l-1}{k-1}. \tag{A.5}$$

A.3 Gaussian Units

The relevant differences between the International System of Units (SI) and Gaussian units are summarised in Table A.1.

Quantity	SI unit	Gaussian unit
charge	Coulomb(C)	Statcoulomb(statC)
mass	Kilogramme(kg)	Gramme(g)
distance	Metre(m)	Centimetre(cm)

Table A.1: *Relevant differences between SI units and Gaussian units*

If two charges Q_1 and Q_2 are separated by distance r then in SI units the magnitude of the force F between them is given by Coulomb's law as

$$F = \frac{1}{4\pi\epsilon_0} \frac{Q_1 Q_2}{r^2}. \tag{A.6}$$

In Gaussian units the same force magnitude is written as

$$F = \frac{Q_1 Q_2}{r^2}. \tag{A.7}$$

A.4 3D FMM Force Calculation

The forces exerted between directly interacting charges can be readily computed from the Coulomb potential. Computing the forces between charges that interact through multipole

and local expansions requires the spatial derivates of local expansions. If the potential field is given by the p-term local expansion centred at the origin

$$\Phi(P) = \sum_{j=0}^{p} \sum_{k=-j}^{j} L_j^k Y_j^k(\theta, \phi) r^j, \tag{A.8}$$

then the electric field $\vec{E}$ at the point $P = (\rho, \alpha, \beta)$ is given by

$$\vec{E}(P) = -\vec{\nabla} \left(\sum_{j=0}^{p} \sum_{k=-j}^{j} L_j^k Y_j^k(\theta, \phi) r^j \right) \Bigg|_{r=\rho,\ \theta=\alpha,\ \phi=\beta}, \tag{A.9}$$

$$\begin{aligned}
&\vec{\nabla} \left(\sum_{j=0}^{p} \sum_{k=-j}^{j} L_j^k Y_j^k(\theta, \phi) r^j \right) = \\
&\hat{\vec{r}} j r^{j-1} Y_j^k(\theta, \phi) \\
&-\hat{\vec{\theta}} \frac{r^{j-1}}{\sin\theta} \sqrt{\frac{(j-|k|)!}{(j+|k|)!}} \exp(ik\theta) \left[j \cos(\theta) P_j^{|k|}(\cos\theta) - (j+|k|) P_{j-1}^{|k|}(\cos\theta) \right] \\
&+\hat{\vec{\phi}} \frac{r^{j-1}}{\sin\theta} \sqrt{\frac{(j-|k|)!}{(j+|k|)!}} P_j^{|k|}(\cos\theta) ik \exp(ik\phi),
\end{aligned} \tag{A.10}$$

where

$$\vec{\nabla} = \hat{\vec{r}} \frac{\partial}{\partial r} + \frac{\hat{\vec{\theta}}}{r} \frac{\partial}{\partial \theta} + \frac{\hat{\vec{\phi}}}{r \sin\theta} \frac{\partial}{\partial \phi}, \tag{A.11}$$

and

$$\hat{\vec{r}} = \begin{pmatrix} r\cos(\phi)\sin(\theta) \\ r\sin(\phi)\sin(\theta) \\ r\cos(\theta) \end{pmatrix}, \quad \hat{\vec{\theta}} = \begin{pmatrix} \cos(\phi)\cos(\theta) \\ \sin(\phi)\cos(\theta) \\ -\sin(\theta) \end{pmatrix}, \quad \hat{\vec{\phi}} = \begin{pmatrix} -\sin(\phi) \\ \cos(\phi) \\ 0 \end{pmatrix}. \tag{A.12}$$

A.5 Balena System Architecture

This work was supported by the University of Bath HPC facility "Balena", system architecture is as outlined in Table A.2

Ivy Bridge Nodes	
CPUs	2x Intel E5-2650v2 8 cores, 2.6Ghz (16 cores per node)
Memory	Quad channel DDR3-1866 MHz (8 channels per node)
Network	Intel True Scale Infiniband (QDR) 40 Gbps

Table A.2: *Balena System Architecture*

A.6 Example LAMMPS Input Script

```
variable    x index 1
variable    y index 1
variable    z index 1
variable    xx equal 20*$x
variable    yy equal 20*$y
variable    zz equal 20*$z
units       lj
atom_style  atomic
lattice     fcc 0.8442
region      box block 0 ${xx} 0 ${yy} 0 ${zz}
create_box  1 box
create_atoms    1 box
mass        1 1.0
velocity    all create 1.44 87287 loop geom
pair_style  lj/cut 2.5
pair_coeff  1 1 1.0 1.0 2.5
neighbor    0.3 bin
neigh_modify    delay 0 every 20 check no
fix     1 all nve
run     100
```

Figure A-1: *LAMMPS Lennard-Jones example script from* ***http://lammps.sandia.gov/inputs/in.lj.txt***

A.7 Example HOOMD-blue Python Input Script

```
from hoomd_script import *

# create 1000 random particles of name A
init.create_random(N=1000, phi_p=0.01, name='A')

# specify Lennard-Jones interactions between particle pairs
lj = pair.lj(r_cut=2.5)
lj.pair_coeff.set('A', 'A', epsilon=1.0, sigma=1.0)

# integrate at constant temperature
all = group.all()
integrate.mode_standard(dt=0.005)
integrate.nvt(group=all, T=1.2, tau=0.5)

# dump an xmle file for the structure information
xml = dump.xml(filename='dump_dcd.xml', vis=True)

# dump a .dcd file for the trajectory
dump.dcd(filename='dump_dcd.dcd', period=100)

# run 10,000 time steps
run(10e3)
```

Figure A-2: *HOOMD-blue Lennard-Jones example Python script from* `http://glotzerlab.engin.umich.edu/hoomd-blue/doc/dump_dcd-example.html`

A.8 CUDA Code Generation

Here we describe the CUDA code generation process using the assumptions and definitions made in Section 3.2.

A.8.1 CUDA Particle Loop

Although modern GPU hardware operates as a wide vector processor GPUs are programmed in a highly threaded shared memory manner, we generate CUDA code to target NVIDIA GPUs. We assign a GPU thread to each particle, for a *Particle Loop* there are no race conditions to access data stored in `ParticleDat` instances, furthermore reading global data is contention free. On GPU hardware our implementation only allows writing to global data using an incremental access descriptor. The code generation system creates code that (1) allocates temporary storage local to each thread for a copy of the global

data and (2) creates code to perform a reduction (addition) across all launched threads on the device. By using temporary storage and inter-thread communication we minimise the use of atomic operations to write values into global memory. The GPU library template is presented in Listing A.1.

Listing A.1: *Template for CUDA based shared library.*

```
<generated_structs>

// CUDA Kernel definition
__global__ void k_<kernel_name> (int _D_N_LOCAL,
    <kernel_parameter_list>)
{
  int _i = threadIdx.x + blockIdx.x*blockDim.x;

  <global_initialisations>

  if (_i<_D_N_LOCAL)
  {
    <kernel_args_creation>

    <kernel_source>
  }

  <global_reductions>
}
// Library function
void <kernel_name>_wrapper(int* _H_BLOCKSIZE, int* _H_THREADSIZE, int
    _H_N_LOCAL, <data_structure_pointers>)
{

  dim3 _B;
  dim3 _T;
  _B.x = _H_BLOCKSIZE[0];
  _B.y = _H_BLOCKSIZE[1];
  _B.z = _H_BLOCKSIZE[2];
  _T.x = _H_THREADSIZE[0];
  _T.y = _H_THREADSIZE[1];
  _T.z = _H_THREADSIZE[2];

  // Kernel call
  k_<kernel_name> <<<_B,_T>>>(_H_N_LOCAL, <kernel_call>);

  checkCudaErrors(cudaDeviceSynchronize());
}
```

The instances of `ParticleDat` generate code that is very similar to the host CPU variant, the major difference is that instances of the generated C structures are created

immediately prior to the kernel code not before the kernel call as in the CPU case. An overview of the code generation process for `ParticleDat` instances is presented in Algorithm 26.

Algorithm 26: Particle Loop code generation for `ParticleDats`

Data: `ParticleDat` instances $D^{(p)}$ and access descriptors $A^{(p)}$

Result: `<generated_structs>`, `<kernel_args_declaration>`, `<data_structure_pointers>`, `<kernel_args_creation>` and `<kernel_call>`

for $m \in \{0, \ldots, m^{(p)}\}$ **do**

Construct identifier `sym` to use for temporary variables (the symbol used in the kernel)

Identify underlying data type `dtype` from $d_m^{(p)}$

Identify number of components `ncomp` from $d_m^{(p)}$

Determine if the `const` qualifier is valid from $a_m^{(p)}$

(1) Create `struct` for `<generated_structs>`:

```
typedef struct {dtype (const) * __restrict__ i;} _sym_t;
```

(2) Create entry for `<kernel_parameter_list>`, a pointer into the global data:

```
dtype (const) * __restrict__ d_sym
```

(3) Create entry for `<data_structure_pointers>`, a pointer: `dtype (const) * __restrict__ sym`

(4) Create entry for `<kernel_args_creation>` using above pointer and struct:

```
_sym_t sym = { d_sym + _i * ncomp };
```

(5) Create entry for `<kernel_call>`, add newly created struct instance to call arguments: `sym`

end

CUDA code that reads global data is generated by creating pointers that map to the correct entry in global memory. To perform reduction operations we create thread local variables which the kernel writes into. The reduction across all GPU threads then occurs in a three stage process, (1) inter-thread communication is applied to reduce all values within a thread warp, (2) thread zero in the warp atomically increments a temporary storage location for the thread block, finally thread zero in the thread block atomically increments the values in global memory. An overview of the CUDA code generation process for global data is presented in Algorithm 27.

Algorithm 27: CUDA Particle Loop code generation for `ScalarArrays` and `GlobalArrays`

Data: `ScalarArray` and `GlobalArray` instances $D^{(s)}$ and access descriptors $A^{(s)}$
Result: `<generated_structs>`, `<kernel_args_declaration>`, `<data_structure_pointers>`, `<kernel_args_creation>` and `<kernel_call>`
for $m \in \{0, \ldots, m^{(s)}\}$ **do**
  Construct identifier `sym` to use for temporary variables (usually the symbol used in the kernel)
  Identify underlying data type `dtype` from $d_m^{(s)}$
  Identify number of components `ncomp` from $d_m^{(p)}$
  Determine if the `const` qualifier is valid from $a_m^{(s)}$
  (1) Create entry for `<kernel_parameter_list>`, a pointer:
  `dtype (const) * d_sym`
  (2) Create entry for `<data_structure_pointers>`: `dtype (const) * sym`
  (3) Create entry for `<kernel_call>`, add pointer to call arguments: `sym`
  if $a_m^{(s)}$ *is read-only* **then**
    (4) Create entry for `<global_initialisations>`, directly map to global data:
    `dtype const * sym = d_sym;`
  end
  if $a_m^{(s)}$ *is increment* **then**
    (4) Create entry for `<global_initialisations>`, create a temporary variable:
    `dtype sym[ncomp] = {0};`
    (5) Create entry for `<global_reductions>`, device-wide reduction from Listing A.2.
  end
end

Listing A.2: *Device-wide reduction for global data accessed with kernel symbol* `sym`*, data type* `dtype` *and number of components* `ncomp`

```
// Reduce across the thread warp with inter-thread communication
for(int _iz = 0; _iz < ncomp; _iz++ ){
    sym[_iz] = warpReduceSum<dtype>(sym[_iz]);
}

// Reduce across the thread block using shared memory
// First create and zero shared memory
__shared__ double _d_red_sym[1];
if (  (int)(threadIdx.x & (warpSize - 1)) == 0){
    for(int _iz = 0; _iz < ncomp; _iz++ ){ _d_red_sym[_iz] = 0; }
} __syncthreads();

// 0^th thread of each warp atomically increments the shared
// elements
if (  (int)(threadIdx.x & (warpSize - 1)) == 0){
    for(int _iz = 0; _iz < ncomp; _iz++ ){
        atomicAdd<dtype>(&_d_red_sym[_iz], sym[_iz]);
    }
} __syncthreads();

// 0^th thread in the thread block increments the globally
// stored elements
if (threadIdx.x == 0){
    for(int _iz = 0; _iz < ncomp; _iz++ ){
        atomicAdd<dtype>(&d_sym[_iz], _d_red_sym[_iz]);
    }
}
```

We extend the CPU *Particle Loop* example to include a `GlobalArray` instance to demonstrate the generated reduction code. The Python source code is presented in Listing A.3 and the output CUDA code is presented in Listing A.4 and A.5. When using multithread shared memory parallelism on CPU architectures we employ the same technique of creating local storage per thread for reduction variables pre kernel launch. Post kernel launch these temporary variables are reduced into the global storage as in the GPU case. However, on CPU architectures multithreading is utilised in a more coarse grain manner and each CPU thread will apply the kernel to multiple particles.

Listing A.3: *Example particle loop creation including incremented global data.*

```
# setup removed for brevity
A.P = ParticleDat(ncomp=2)
S = ScalarArray(ncomp=2)
G = GlobalArray(ncomp=1)

particle_loop = ParticleLoop(
    kernel=Kernel(
        name='plexample',
        code = """
        P.i[0] = S[0];
        P.i[1] = S[1];
        G[0]++;
        """
    ),
    dat_dict={
        'P': A.P(INC),
        'S': S(READ),
        'G': G(INC)
    }
)
```

Listing A.4: *Example generated CUDA particle loop from input A.3, Part I.*

```
// Structs generated per ParticleDat
typedef struct { double *__restrict__ i; } _P_t;

// Kernel device function (CUDA kernel)
__global__ void k_plexample(int const _D_N_LOCAL, double *__restrict__
    d_G, double *__restrict__ d_P, double const *__restrict__ d_S)
{
  int const _i = threadIdx.x + blockIdx.x*blockDim.x;

  // Global data access/initialisation
  double G[1] = {0};
  double const *S = d_S;

  if (_i<_D_N_LOCAL) {
    // ParticleDat structs instances
    _P_t P = { d_P+_i*2};

    // User supplied kernel
    P.i[0] = S[0];
    P.i[1] = S[1];
    G[0]++;
  }

  // global data reductions
  for(int _iz = 0; _iz < 1; _iz++ ){G[_iz] =
      warpReduceSumDouble(G[_iz]); }
  __shared__ double _d_red_G[1];
  if (  (int)(threadIdx.x & (warpSize - 1)) == 0){
    for(int _iz = 0; _iz < 1; _iz++ ){
    _d_red_G[_iz] = 0;
  }} __syncthreads();
  if (  (int)(threadIdx.x & (warpSize - 1)) == 0){
    for(int _iz = 0; _iz < 1; _iz++ ){
    atomicAddDouble(&_d_red_G[_iz], G[_iz]);
  }} __syncthreads();
  if (threadIdx.x == 0){
    for(int _iz = 0; _iz < 1; _iz++ ){
    atomicAddDouble(&d_G[_iz], _d_red_G[_iz]);
  }}
}
```

Listing A.5: *Example generated CUDA particle loop from input A.3, Part II.*

```
// Library function
void plexample_wrapper(int const *__restrict__ _H_BLOCKSIZE, int const
    *__restrict__ _H_THREADSIZE, int const _H_N_LOCAL, double
    *__restrict__ G, double *__restrict__ P, double const *__restrict__
    S)
{
  dim3 _B;
  dim3 _T;
  _B.x = _H_BLOCKSIZE[0];
  _B.y = _H_BLOCKSIZE[1];
  _B.z = _H_BLOCKSIZE[2];
  _T.x = _H_THREADSIZE[0];
  _T.y = _H_THREADSIZE[1];
  _T.z = _H_THREADSIZE[2];
  k_plexample<<<_B,_T>>>(_H_N_LOCAL,G,P,S);
  checkCudaErrors(cudaDeviceSynchronize());
}
```

A.8.2 CUDA Local Particle Pair Loop

On GPU architectures the neighbour list is constructed as a neighbour matrix as described in Section 3.1.5. The GPU particle pair loop template in Listing A.1 is amended in Listing A.6 to include code that extracts neighbours from a neighbour list.

Listing A.6: *Template for CUDA based Local Particle Pair Loop.*

```
<generated_structs>

// CUDA Kernel definition
__global__ void k_<kernel_name> (int _D_N_LOCAL, int *_D_NMATRIX,
    <kernel_parameter_list>)
{
  int _i = threadIdx.x + blockIdx.x*blockDim.x;

  <global_initialisations>

  if (_i<_D_N_LOCAL)
  {
    // loop over entries in neighbour matrix
        // first row contains the number of neighbours
    for (int _k=1; _k<=_D_NMATRIX[_i]; _k++)
    {
      int _j = _D_NMATRIX[_i + _D_N_LOCAL * _k ];

      <kernel_args_creation>

      <kernel_source>
    }
  }

  <global_reductions>
}
// Library function
void <kernel_name>_wrapper(int* _H_BLOCKSIZE, int* _H_THREADSIZE, int
    _H_N_LOCAL, int *_D_NMATRIX, <data_structure_pointers>)
{

  dim3 _B;
  dim3 _T;
  _B.x = _H_BLOCKSIZE[0];
  _B.y = _H_BLOCKSIZE[1];
  _B.z = _H_BLOCKSIZE[2];
  _T.x = _H_THREADSIZE[0];
  _T.y = _H_THREADSIZE[1];
  _T.z = _H_THREADSIZE[2];

  // Kernel call
  k_<kernel_name> <<<_B,_T>>>(_H_N_LOCAL, _D_NMATRIX, <kernel_call>);
  checkCudaErrors(cudaDeviceSynchronize());
}
```

The Python source in Listing 3.7 generates the CUDA source code in Listing A.7 and

A.8.

Listing A.7: *Generated CUDA code to count the neighbours of each particle within a radius 2 using a GPU, part I.*

```
// Structs generated per ParticleDat
typedef struct
{
  double const *i;
  double const *j;
}  _P_t;
typedef struct
{
  int *i;
  int *j;
}  _NC_t;

// Kernel function
__global__ void k_n_count(int const _D_N_LOCAL, int *_D_NMATRIX, double
    const * d_P, int *d_NC)
{
  int const _i = threadIdx.x + blockIdx.x*blockDim.x;

  if (_i<_D_N_LOCAL)
  {
    for (int _k=1; _k<=_D_NMATRIX[_i]; _k++)
    {
      int const _j = _D_NMATRIX[_i + _D_N_LOCAL * _k ];

      _P_t P = { d_P+_i*3, d_P+_j*3};
      _NC_t NC = { d_NC+_i*1, d_NC+_j*1};

      double r0 = P.i[0] - P.j[0];
      double r1 = P.i[1] - P.j[1];
      double r2 = P.i[2] - P.j[2];
      if ((r0*r0 + r1*r1 + r2*r2) < 4.0){
          NC.i[0] += 1;
      }

    }
  }
}
```

Listing A.8: *Generated CUDA code to count the neighbours of each particle within a radius 2 using a GPU, part II.*

```
// Library function
void n_count_wrapper(int * _H_BLOCKSIZE, int * _H_THREADSIZE, int
    _H_N_LOCAL, int * _D_NMATRIX, double const * P, int * NC)
{
  dim3 _B;
  dim3 _T;
  _B.x = _H_BLOCKSIZE[0];
  _B.y = _H_BLOCKSIZE[1];
  _B.z = _H_BLOCKSIZE[2];
  _T.x = _H_THREADSIZE[0];
  _T.y = _H_THREADSIZE[1];
  _T.z = _H_THREADSIZE[2];
  k_n_count<<<_B,_T>>>(_H_N_LOCAL,_D_NMATRIX,P,NC);
  checkCudaErrors(cudaDeviceSynchronize());
}
```

A.9 Convergence Characteristics Of Ewald Summation

With periodic boundary conditions the simulated system consists of a primary image surrounded by a lattice of periodic images. In the 3D FMM we constructed the multipole expansion $\Phi_{0,0}$ which describes the potential induced by any image of the simulation cell, with expansion coefficients M_n^m the potential at the centre of the primary image from periodic image $\vec{\nu}$ is

$$\varphi_{\vec{\nu}}(\vec{0}) = \sum_{n=0}^{\infty} \sum_{m=-n}^{n} \frac{M_n^m}{r_{\vec{\nu}}^{n+1}} Y_n^m(\theta_{\vec{\nu}}, \phi_{\vec{\nu}}). \tag{A.13}$$

where $(r_{\vec{\nu}}, \theta_{\vec{\nu}}, \phi_{\vec{\nu}})$ is the spherical coordinate vector to the centre of image $\vec{\nu}$. In Section 4.3.2 we described the method by Amisaki [7] to compute the matrix R such that

$$R_n^m = \sum_{\vec{\nu}} \frac{Y_n^m(\theta_{\vec{\nu}}, \phi_{\vec{\nu}})}{r_{\vec{\nu}}^{n+1}}. \tag{A.14}$$

This summation method by Amisaki assigns physically sensible values to summations of the form

$$\varphi_\alpha = \sum_{\vec{\nu} \in \mathbb{Z}^3 \setminus \vec{0}} \frac{1}{|\vec{r}_{\vec{\nu}}|^\alpha} \tag{A.15}$$

where $\alpha \in \{1, 2, 3\}$, which allows periodic boundary conditions in the 3D FMM. In general, for $\alpha < 4$ these summations are conditionally convergent as described by the Riemann rearrangement theorem. To study the behaviour of these summations in the Ewald method we consider the $\alpha = 1$ and $\alpha = 2$ cases which correspond to the monopole and dipole charge distributions, the $\alpha = 3$ quadrupole case could also be of interest.

The monopole case is zero by the assumption of charge neutrality. This assumption allows the long-range component of the Ewald method to neglect the $\vec{k} = \vec{0}$ term and is physically sensible. To investigate the dipole term we investigate the long-range energy contribution in the $\alpha = 2$ case by considering a charge density which is a point dipole.

Consider two charges $q_1 = -q$ and $q_2 = +q$ at positions $\vec{r}_1 = (-d/2, 0, 0)$ and $\vec{r}_2 = (d/2, 0, 0)$ respectively, as in Figure A-3. The dipole moment of these two charges has magnitude $p = qd$ and is parallel to the x-axis. A point dipole at the origin is formed in the limit as $d \to 0$ with p constant.

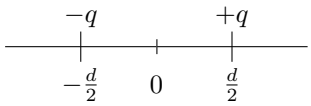

Figure A-3: *Two charges of strength q separated by distance d aligned parallel to the x-axis.*

In the Ewald method the long-range contribution to the electrostatic energy is given by

$$\phi^{(\mathrm{lr})} = \frac{1}{2V} \sum_{\substack{\vec{k} \neq \vec{0} \\ |\vec{k}| < k_c}} \frac{4\pi}{|\vec{k}|^2} \exp\left(\frac{-|\vec{k}|^2}{4\alpha}\right) |\bar{\rho}(\vec{k})|^2, \tag{A.16}$$

where

$$\bar{\rho}(\vec{k}) = \sum_{j=1}^{N} q_j \exp(i\vec{k} \cdot \vec{r}_j). \tag{A.17}$$

For our two charge system the real part of $\bar{\rho}(\vec{k})$ is

$$\begin{aligned} \mathrm{Re}\left(\bar{\rho}(\vec{k})\right) &= -q\cos(\vec{k}_x(-d/2)) + q\cos(\vec{k}_x d/2), & \text{(A.18)} \\ &= 0, & \text{(A.19)} \end{aligned}$$

and the imaginary part is

$$\begin{aligned} \mathrm{Im}\left(\bar{\rho}(\vec{k})\right) &= -q\sin(\vec{k}_x(-d/2)) + q\sin(\vec{k}_x d/2), & \text{(A.20)} \\ &= 2q\sin(\vec{k}_x d/2), & \text{(A.21)} \\ &= \frac{2p}{d}\sin(\vec{k}_x d/2). & \text{(A.22)} \end{aligned}$$

We are interested in the contribution of a point dipole which is given by

$$\begin{aligned} \lim_{d\to 0}\left(\mathrm{Im}\left(\bar{\rho}(\vec{k})\right)\right) &= \lim_{d\to 0}\left(\frac{2p}{d}\sin(\vec{k}_x d/2)\right), & \text{(A.23)} \\ &= p\vec{k}_x. & \text{(A.24)} \end{aligned}$$

Hence the long-range energy $\phi^{(\mathrm{lr})}$ in the limit $d \to 0$ is

$$\phi^{(\mathrm{lr})} = \frac{1}{2V} \sum_{\substack{\vec{k} \neq \vec{0} \\ |\vec{k}| < k_c}} \frac{4\pi}{|\vec{k}|^2} \exp\left(\frac{-|\vec{k}|^2}{4\alpha}\right) p^2 |\vec{k}_x|^2, \tag{A.25}$$

$$\leq \frac{2\pi p^2}{V} \sum_{\substack{\vec{k} \neq \vec{0} \\ |\vec{k}| < k_c}} \exp\left(\frac{-|\vec{k}|^2}{4\alpha}\right), \tag{A.26}$$

which is a convergent summation. To compare Equation (A.25) with the system energy seen in practice from our Ewald implementation presented in section 5.1 we constructed the system illustrated in Figure A-3. In Figure A-4 we plot the long-range contribution to the system energy for varying charge separation distances d and plot the predicted value for $d \to 0$. Although the long-range contribution to the energy converges the short-range contribution diverges as $\mathrm{erfc}(\sqrt{\alpha}r)/r \to \infty$ as $r \to 0$.

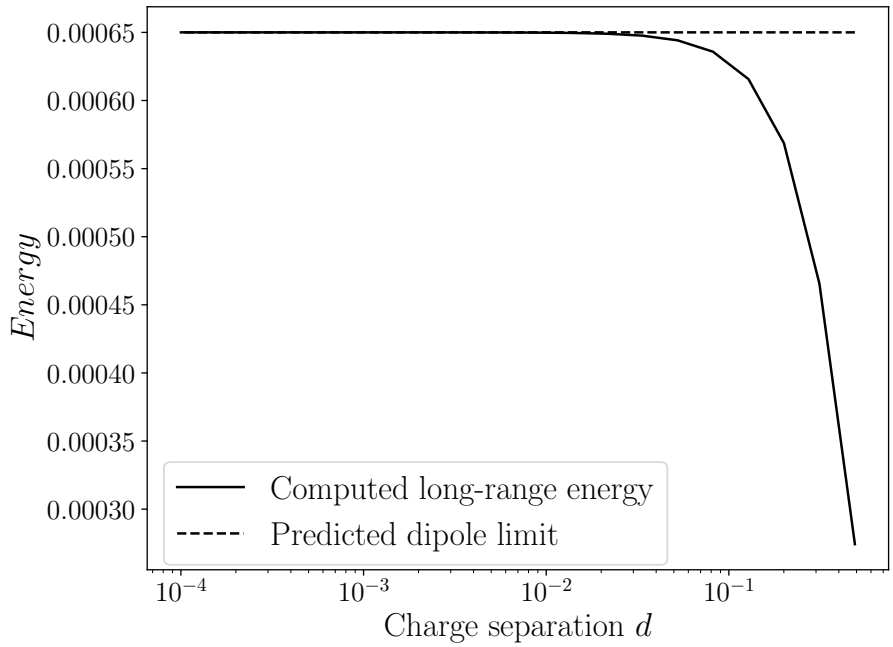

Figure A-4: *Long-range energy contribution of an approximate dipole system constructed from two charges separated by a distance d. Predicted long-range energy is plotted in dashed black, computed energy is plotted in solid black. Values are plotted for Gaussian width $\alpha = 1.0$ and reciprocal cutoff $k_c = 200$.*

The long-range potential $\phi^{(\mathrm{lr})}$ at a point $\vec{b}$ is

$$\phi^{(\mathrm{lr})}(\vec{b}) = \frac{1}{V} \sum_{\substack{\vec{k} \neq \vec{0} \\ |\vec{k}| < k_c}} \frac{4\pi}{|\vec{k}|^2} \exp\left(\frac{-|\vec{k}|^2}{4\alpha}\right) \exp(i\vec{k} \cdot \vec{b}) \rho(\vec{k}), \tag{A.27}$$

where for a dipole charge distribution

$$\rho(\vec{k}) = \sum_{j=1}^{N} q_j \exp(-i\vec{k} \cdot \vec{r}_j), \tag{A.28}$$

$$= -p\vec{k}_x. \tag{A.29}$$

The long-range potential at the origin induced by a periodic lattice of dipoles is

$$\phi^{(\mathrm{lr})}(\vec{0}) = \frac{1}{V} \sum_{\substack{\vec{k}\neq\vec{0} \\ |\vec{k}|<k_c}} \frac{4\pi}{|\vec{k}|^2} \exp\left(\frac{-|\vec{k}|^2}{4\alpha}\right)(-p\vec{k}_x), \tag{A.30}$$

$$= 0 \quad \text{for finite } k_c. \tag{A.31}$$

If we consider a point $\vec{b} \neq \vec{0}$ then for the long-range potential

$$|\phi^{(\mathrm{lr})}(\vec{b})| = \frac{1}{V} \sum_{\substack{\vec{k}\neq\vec{0} \\ |\vec{k}|<k_c}} \left| \frac{4\pi}{|\vec{k}|^2} \exp\left(\frac{-|\vec{k}|^2}{4\alpha}\right) \exp(i\vec{k} \cdot \vec{b})(-p\vec{k}_x) \right|, \tag{A.32}$$

$$\leq C_1 \sum_{\substack{\vec{k}\neq\vec{0} \\ |\vec{k}|<k_c}} \frac{1}{|\vec{k}|^2} \exp\left(\frac{-|\vec{k}|^2}{4\alpha}\right) |\vec{k}_x|, \tag{A.33}$$

$$\leq C_1 \sum_{\substack{\vec{k}\neq\vec{0} \\ |\vec{k}|<k_c}} \exp\left(\frac{-|\vec{k}|^2}{4\alpha}\right), \tag{A.34}$$

$$\leq C_2 \int_{k=0}^{\infty} \exp\left(\frac{-|\vec{k}|^2}{4\alpha}\right) = C_2\sqrt{\alpha\pi}. \tag{A.35}$$

We conclude that the Fourier Transform approach defines an ordering of the summation in Equation (A.15) that is convergent for all reciprocal cutoffs k_c. By defining an ordering of the summation the Ewald method chooses a value for these conditionally convergent summations. We see in the case of a dipole only system the potential at the origin does make physical sense, we do not discuss higher order moments.

BIBLIOGRAPHY

[1] F. F. Abraham, R. Walkup, H. Gao, M. Duchaineau, T. Diaz De La Rubia, and M. Seager, *Simulating materials failure by using up to one billion atoms and the world's fastest computer: Work-hardening*, Proceedings of the National Academy of Sciences, 99 (2002), pp. 5783–5787.

[2] M. P. Allen and D. J. Tildesley, *Computer Simulation of Liquids (Oxford Science Publications)*, Oxford science publications, Clarendon Press, 1989.

[3] N. Allsopp, G. Ruocco, and A. Fratalocchi, *Molecular dynamics beyonds the limits: Massive scaling on 72 racks of a BlueGene/P and supercooled glass dynamics of a 1 billion particles system*, Journal of Computational Physics, 231 (2012), pp. 3432 – 3445.

[4] M. S. Alnæs, J. Blechta, J. Hake, A. Johansson, B. Kehlet, A. Logg, C. Richardson, J. Ring, M. E. Rognes, and G. N. Wells, *The FEniCS Project Version 1.5*, Archive of Numerical Software, 3 (2015).

[5] M. S. Alnæs, A. Logg, K. B. Ølgaard, M. E. Rognes, and G. N. Wells, *Unified Form Language: A domain-specific language for weak formulations of partial differential equations*, CoRR, abs/1211.4047 (2012).

[6] G. M. Amdahl, *Validity of the Single Processor Approach to Achieving Large Scale Computing Capabilities*, in Proceedings of the April 18-20, 1967, Spring Joint Computer Conference, AFIPS '67 (Spring), New York, NY, USA, 1967, ACM, pp. 483–485.

[7] T. Amisaki, *Precise and efficient Ewald summation for periodic fast multipole method*, Journal of Computational Chemistry, 21, pp. 1075–1087.

[8] H. C. Andersen, *Molecular dynamics simulations at constant pressure and/or temperature*, The Journal of Chemical Physics, 72 (1980), pp. 2384–2393.

[9] J. A. Anderson, C. D. Lorenz, and A. Travesset, *General purpose molecular dynamics simulations fully implemented on graphics processing units*, Journal of Computational Physics, 227 (2008), pp. 5342 – 5359.

[10] S. Balay, S. Abhyankar, M. F. Adams, J. Brown, P. Brune, K. Buschelman, L. Dalcin, V. Eijkhout, W. D. Gropp, D. Kaushik, M. G. Knepley, L. C. McInnes, K. Rupp, B. F. Smith, S. Zampini, H. Zhang, and H. Zhang, *PETSc Web page*. `http://www.mcs.anl.gov/petsc`, 2016. [Online; accessed 26/05/2016].

[11] L. Boltzmann, *Weitere studien über das Wärmegleichgewicht unter Gasmolekülen*, in Sitzungsberichte der Kaiserlichen Akademie der Wissenschaften in Wien, Mathematisch-Naturwissenschaftliche Classe, vol. 66, 1872, pp. 275–370.

[12] ——, *Über die Beziehung zwischen dem zweiten Hauptsatz der mechanischen Wärmetheorie und der Wahrscheinlichkeitsrechnung respektive den Sätzen über das Wärmegleichgewicht.*, in Sitzungsberichte der Kaiserlichen Akademie der Wissenschaften in Wien, Mathematisch-Naturwissenschaftliche Classe, vol. 76, 1877, pp. 373–435.

[13] Cavium and ARM, *ThunderX2 ARM processor*. `https://www.cavium.com/product-thunderx2-arm-processors.html`.

[14] CCP5, *DL_POLY_4 TEST01*. `ftp://ftp.dl.ac.uk/ccp5/DL_POLY/DL_POLY_4.0/DATA/`, 2018. [Online; accessed 01/04/2018].

[15] L. Dalcín, R. Paz, M. Storti, and J. D'Elía, *MPI for Python: Performance improvements and MPI-2 extensions*, Journal of Parallel and Distributed Computing, 68 (2008), pp. 655 – 662.

[16] T. Darden, D. York, and L. Pedersen, *Particle mesh Ewald: An $N \cdot \log(N)$ method for Ewald sums in large systems*, The Journal of Chemical Physics, 98 (1993), pp. 10089–10092.

[17] D.C.Rapaport, *Enhanced molecular dynamics performance with a programmable graphics processor*, Computer Physics Communications, 182 (2011), pp. 926–934.

[18] M. Deserno and C. Holm, *How to mesh up Ewald sums. I. A theoretical and numerical comparison of various particle mesh routines*, The Journal of Chemical Physics, 109 (1998), pp. 7678–7693.

[19] S. Duane, A. Kennedy, B. J. Pendleton, and D. Roweth, *Hybrid Monte Carlo*, Physics Letters B, 195 (1987), pp. 216 – 222.

[20] P. Eastman and V. Pande, *Accelerating Development and Execution Speed with Just-in-Time GPU Code Generation*, in GPU Computing Gems, M. Kaufmann, ed., vol. 2, pp. 399–407.

[21] ——, *Openmm: A hardware-independent framework for molecular simulations*, Computing in Science Engineering, 12 (2010), pp. 34–39.

[22] U. Essmann, L. Perera, M. L. Berkowitz, T. Darden, H. Lee, and L. G. Pedersen, *A smooth particle mesh Ewald method*, The Journal of Chemical Physics, 103 (1995), pp. 8577–8593.

[23] P. P. Ewald, *Die Berechnung optischer und elektrostatischer Gitterpotentiale*, Annalen der Physik, 369 (1921), pp. 253–287.

[24] D. Frenkel and B. Smit, in Understanding Molecular Simulation (Second Edition) , Academic Press, 2002, p. 64.

[25] ——, in Understanding Molecular Simulation (Second Edition) , Academic Press, 2002, p. 12.

[26] ——, in Understanding Molecular Simulation (Second Edition) , Academic Press, 2002, p. 291.

[27] J. G. Gay and B. J. Berne, *Modification of the overlap potential to mimic a linear site-site potential*, The Journal of Chemical Physics, 74 (1981), pp. 3316–3319.

[28] Z. Gimbutas and L. Greengard, *A fast and stable method for rotating spherical harmonic expansions*, Journal of Computational Physics, 228 (2009), pp. 5621 – 5627.

[29] M. S. Green, *Markov Random Processes and the Statistical Mechanics of Time-Dependent Phenomena. II. Irreversible Processes in Fluids*, The Journal of Chemical Physics, 22 (1954), pp. 398–413.

[30] L. Greengard, *The Rapid Evaluation of Potential Fields in Particle Systems*, PhD thesis, Yale University, 1987.

[31] L. Greengard and V. Rokhlin, *A fast algorithm for particle simulations*, Journal of Computational Physics, 73 (1987), pp. 325 – 348.

[32] ——, *A new version of the fast multipole method for the laplace equation in three dimensions*, Acta Numerica, 6 (1997), pp. 229 – 269.

[33] GW4, *Isambard*. `http://gw4.ac.uk/isambard/`.

[34] W. K. Hastings, *Monte Carlo Sampling Methods Using Markov Chains and Their Applications*, Biometrika, 57.1 (1970), pp. 97–109.

[35] R. Hockney and J. Eastwood, *Computer Simulation Using Particles*, 1988.

[36] M. Homolya, L. Mitchell, F. Luporini, and D. Ham, *TSFC: A Structure-Preserving Form Compiler*, SIAM Journal on Scientific Computing, 40 (2018), pp. C401–C428.

[37] J. D. Honeycutt and H. C. Andersen, *Molecular dynamics study of melting and freezing of small Lennard-Jones clusters*, The Journal of Physical Chemistry, 91 (1987), pp. 4950–4963.

[38] W. G. Hoover, *Canonical dynamics: Equilibrium phase-space distributions*, Phys. Rev. A, 31 (1985), pp. 1695–1697.

[39] ——, *Constant-pressure equations of motion*, Phys. Rev. A, 34 (1986), pp. 2499–2500.

[40] Intel Corporation. `http://ark.intel.com/products/75269/Intel-Xeon-Processor-E5-2650-v2-20M-Cache-2_60-GHz`, 2013. [Online; accessed 14/09/2017].

[41] I.T.Todorov, W.Smith, K.Trachenko, and M.T.Dove, *DL_POLY*, Journal of Materials Chemistry, 16 (2006), pp. 1911–1918.

[42] A. V. Ivlev, J. Bartnick, M. Heinen, C.-R. Du, V. Nosenko, and H. Löwen, *Statistical Mechanics where Newton's Third Law is Broken*, Phys. Rev. X, 5 (2015), p. 011035.

[43] J. Jackson, *Classical Electrodynamics*, John Wiley & Sons, 3 ed., 7 1998.

[44] K. Kadau, T. Germann, and P. S. Lomdahl, *Molecular Dynamics Comes of Age:. 320 Billion Atom Simulation on BlueGene/L*, 17 (2006), pp. 1755–1761.

[45] J. Kolafa and J. W. Perram, *Cutoff Errors in the Ewald Summation Formulae for Point Charge Systems*, Molecular Simulation, 9 (1992), pp. 351–368.

[46] R. Kubo, *Statistical-Mechanical Theory of Irreversible Processes. I. General Theory and Simple Applications to Magnetic and Conduction Problems*, Journal of the Physical Society of Japan, 12 (1957), pp. 570–586.

[47] J. E. Lennard-Jones, *Cohesion*, Proceedings of the Physical Society, 43 (1931), p. 461.

[48] A. Malins, S. R. Williams, J. Eggers, and C. P. Royall, *Identification of structure in condensed matter with the topological cluster classification*, The Journal of Chemical Physics, 139 (2013).

[49] J. C. Maxwell, *II. Illustrations of the dynamical theory of gases*, The London, Edinburgh, and Dublin Philosophical Magazine and Journal of Science, 20 (1860), pp. 21–37.

[50] ——, *V. Illustrations of the dynamical theory of gases. Part I. On the motions and collisions of perfectly elastic spheres*, The London, Edinburgh, and Dublin Philosophical Magazine and Journal of Science, 19 (1860), pp. 19–32.

[51] W. Mickel, S. C. Kapfer, G. E. Schröder-Turk, and K. Mecke, *Shortcomings of the bond orientational order parameters for the analysis of disordered particulate matter*, The Journal of Chemical Physics, 138 (2013), p. 044501.

[52] R. M. Neal, *Handbook of Markov Chain Monte Carlo*, in Handbooks of Modern Statistical Methods, CRC Press, 2011, ch. MCMC Using Hamiltonian Dynamics.

[53] I. Newton, *AXIOMATA SIVE LEGES MOTUS*, in Principia Mathematica, 1687.

[54] S. Nosé, *A molecular dynamics method for simulations in the canonical ensemble*, Molecular Physics, 52 (1984), pp. 255–268.

[55] ——, *A unified formulation of the constant temperature molecular dynamics methods*, The Journal of Chemical Physics, 81 (1984), pp. 511–519.

[56] NVIDIA Corporation. `http://www.nvidia.co.uk/content/PDF/kepler/Tesla-K20X-BD-06397-001-v05.pdf`, November 2012. [Online; accessed 14/09/2017].

[57] ——, *NVIDIA Tesla P100*. `http://www.nvidia.com/object/tesla-p100.html`, 2016. [Online; accessed 03/06/2016].

[58] ——. `http://www.nvidia.com/content/tesla/pdf/nvidia-tesla-kepler-family-datasheet.pdf`, 2017.

[59] U. of Michigan et al, *HOOMD-blue web page*. `http://glotzerlab.engin.umich.edu/hoomd-blue/`, 2016. [Online; accessed 25/05/2016].

[60] S. Páll, M. J. Abraham, C. Kutzner, B. Hess, and E. Lindahl, *Tackling exascale software challenges in molecular dynamics simulations with GROMACS*, CoRR, abs/1506.00716 (2015).

[61] D. A. Pearlman, D. A. Case, J. W. Caldwell, W. S. Ross, T. E. Cheatham, S. DeBolt, D. Ferguson, G. Seibel, and P. Kollman, *AMBER, a package of computer programs for applying molecular mechanics, normal mode analysis, molecular dynamics and free energy calculations to simulate the structural and energetic properties of molecules*, Computer Physics Communications, 91 (1995), pp. 1 – 41.

[62] J. C. Phillips, R. Braun, W. Wang, J. Gumbart, E. Tajkhorshid, E. Villa, C. Chipot, R. D. Skeel, L. Kalé, and K. Schulten, *Scalable molecular dynamics with NAMD*, Journal of Computational Chemistry, 26 (2005), pp. 1781–1802.

[63] S. Plimpton, *Fast Parallel Algorithms for Short-Range Molecular Dynamics*, Journal of Computational Physics, 117 (1995), pp. 1 – 19.

[64] ——, *LAMMPS gpu package documentation*. `http://lammps.sandia.gov/doc/accelerate_gpu.html`, 2016. [Online; accessed 03/06/2016].

[65] ——, *Lennard-Jones liquid benchmark*. `http://lammps.sandia.gov/bench.html#lj`, 2016. [Online; accessed 03/06/2016].

[66] Python Software Foundation, *Python*. `https://www.python.org/`, 2018.

[67] F. Rathgeber, D. A. Ham, L. Mitchell, M. Lange, F. Luporini, A. T. T. Mcrae, G.-T. Bercea, G. R. Markall, and P. H. J. Kelly, *Firedrake: Automating the Finite Element Method by Composing Abstractions*, ACM Trans. Math. Softw., 43 (2016), pp. 24:1–24:27.

[68] F. Rathgeber, G. Markall, L. Mitchell, N. Loriant, D. Ham, C. Bertolli, and P. Kelly, *PyOP2: A High-Level Framework for Performance-Portable Simulations on Unstructured Meshes*, in High Performance Computing, Networking, Storage and Analysis (SCC), 2012 SC Companion:, 2012, pp. 1116–1123.

[69] ——, *PyOP2 Documentation*. `http://op2.github.io/PyOP2/index.html/`, 2016. [Online; accessed 25/05/2016].

[70] C. P. Royall and S. R. Williams, *The role of local structure in dynamical arrest*, Physics Reports, 560 (2015), pp. 1 – 75.

[71] W. R. Saunders. `https://doi.org/10.5281/zenodo.496142`, 2017.

[72] ——. `https://doi.org/10.5281/zenodo.496147`, 2017.

[73] W. R. Saunders, J. Grant, and E. H. Müller, *A domain specific language for performance portable molecular dynamics algorithms*, Computer Physics Communications, 224 (2018), pp. 119 – 135.

[74] ——, *Long Range Forces in a Performance Portable Molecular Dynamics Framework*, in Parallel Computing is Everywhere, 2018, pp. 37 – 46.

[75] P. J. Steinhardt, D. R. Nelson, and M. Ronchetti, *Bond-orientational order in liquids and glasses*, Phys. Rev. B, 28 (1983), pp. 784–805.

[76] A. Stukowski, *Structure identification methods for atomistic simulations of crystalline materials*, Modelling and Simulation in Materials Science and Engineering, 20 (2012), p. 045021.

[77] W. C. Swope, H. C. Andersen, P. H. Berens, and K. R. Wilson, *A computer simulation method for the calculation of equilibrium constants for the formation of physical clusters of molecules: Application to small water clusters*, The Journal of Chemical Physics, 76 (1982), pp. 637–649.

[78] S. van der Walt, S. C. Colbert, and G. Varoquaux, *The NumPy Array: A Structure for Efficient Numerical Computation*, Computing in Science and Engineering, 13 (2011), pp. 22–30.

[79] L. Verlet, *Computer "Experiments" on Classical Fluids. I. Thermodynamical Properties of Lennard-Jones Molecules*, Phys. Rev., 159 (1967), pp. 98–103.

[80] E. Wigner, *Gruppentheorie und ihre Anwendungen auf die Quantenmechanik der Atomspektren. Braunschweig: Vieweg Verlag*, Friedr. Vieweg & Son, 1931.